Seminar

Sicherheitskritische Systeme

7. und 8. Oktober 2004

Universität Siegen

Inhalt

Beispiele für SkS u.a. Stromausfall in USA, Bodenseeunglück, TCAS System, Begriffsklärung
Vortragender: Henning Westerholt

Qualitätsmanagement für SkS und Zertifizierung
Vortragender: Christian Dörner

Risiko- und Gefahrenanalysen und deren Berücksichtigung beim Entwurf von SkS
Vortragender: Kelen-Yo Rodrigue

Untersuchung der Programmiersprache ADA zur Implementierung sicherheitskritischer Systeme
Vortragender: Peter Sakwerda

Testen, Validierung, Verifikation von SkS
Vortragender: Chandra Jaya

Modelchecking: Eine Einführung
Vortragender: Yuguo Sun

UML 2.0: Unified Modelling Language inklusive Semantik?
Vortragender: Christoph Hardt

Codegenerierung für UML 2.0 Modelle
Vortragender: Beyhan Gögeli

Real time Linux: Ein Überblick
Vortragender: Marcus Klein

Real Time Java: Ein Überblick
Vortragender: Frank Köther

WindowsCE als Real Time Betriebssystem?!
Vortragender: Henning Peuser

Begriffserklärungen und Beispiele für Sicherheitskritischer Systeme

Seminararbeit
im Fach Informatik
im Rahmen des Seminars "**Sicherheitskritische Systeme**"
an der

Universität Siegen, Fachgruppe für Praktische Informatik

eingereicht bei
Dr. Jörg Niere

vorgelegt von
Henning Westerholt

Sommersemester 2004
Siegen, April 2004

Inhaltsverzeichnis

1 Begrifflichkeiten Sicherheitskritischer Systeme 3
- 1.1 Allgemeine Definitionen . . . 3
  - 1.1.1 Was sind Sicherheitskritische Systeme? . . . 3
  - 1.1.2 Safety vs. Security . . . 3
  - 1.1.3 Was ist Sicherheit? . . . 3
  - 1.1.4 Formen von SkS . . . 3
  - 1.1.5 Unterscheidung Hochintegrierte Systeme - SkS . . . 4
- 1.2 Risiken . . . 4
- 1.3 Sicherheitskriterien . . . 4
  - 1.3.1 Systemanforderungen . . . 4
  - 1.3.2 Konflikte zwischen Systemanforderungen . . . 5
  - 1.3.3 Sicherheitsanforderungen . . . 5
- 1.4 Fehler . . . 6
  - 1.4.1 Fehler, Fehlfunktion und Systemausfall . . . 6
  - 1.4.2 Unterscheidung über die Fehlerart . . . 6
  - 1.4.3 Unterscheidung über die Fehlerdauer . . . 6
  - 1.4.4 Sonstige Unterscheidungen . . . 7
- 1.5 Fehlervermeidung . . . 7
- 1.6 Fehlerbehebung . . . 7
- 1.7 Fehlerkennung . . . 8
- 1.8 Fehlertoleranz . . . 8

2 Der Stromausfall am 14. August 2003 in den USA 8
- 2.1 Einleitung . . . 8
- 2.2 Gründe . . . 9
  - 2.2.1 Mangelhaftes Systemverständnis und Überlastung des Netzes . 9
  - 2.2.2 Ungenügendes Situationsbewusstsein und technische Probleme bei FE . . . 9
  - 2.2.3 Unzureichendes Beschneiden der Bäume . . . 10
  - 2.2.4 Falsche oder keine Echtzeitüberwachungsdaten und mangelhafte Zusammenarbeit . . . 10
- 2.3 Weiterer Verlauf der Ereignisse . . . 10
- 2.4 Anmerkungen . . . 11

3 Der Flugzeugabsturz bei Überlingen am Bodensee 12
- 3.1 Einleitung . . . 12
- 3.2 Das TCAS System . . . 12
  - 3.2.1 Hintergrund . . . 12
  - 3.2.2 Einstufung . . . 12
  - 3.2.3 Technische Funktionsweise . . . 13
  - 3.2.4 Mensch - Maschine Interface . . . 13
  - 3.2.5 Probleme . . . 13
- 3.3 Gründe . . . 14
  - 3.3.1 Technische Mängel bei der Flugsicherung . . . 14
  - 3.3.2 Stress und Überlastung . . . 14
  - 3.3.3 Probleme durch TCAS . . . 14
  - 3.3.4 Formelle Probleme . . . 14
- 3.4 Anmerkungen und Konsequenzen . . . 15

1 Begrifflichkeiten Sicherheitskritischer Systeme

Exakte Definitionen sind für grundlegend für den Entwurf und das Design von SkS[1]. Dieser Teil dieser Arbeit soll helfen, Unklarheiten die bei der Verwendung der oft auch umgangssprachlich genutzten Begriffe öfters auftreten, auszuräumen. Alle Definitionen in diesem Abschnitt sind an die Veröffentlichungen von [Storey] und [Giese] angelehnt.

1.1 Allgemeine Definitionen

1.1.1 Was sind Sicherheitskritische Systeme?

Als Sicherheitskritische Systeme, oder auch Sicherheitsrelevante Systeme bezeichnet man Systeme, die die Sicherheit einer Anlage[2] garantieren bzw. sie unterstützen.

1.1.2 Safety vs. Security

Im Englischen unterscheidet man zwischen Safety und Security. Leider gibt es in der Deutschen Sprache keine Unterscheidung dieser Begriffe. Ich beschäftige mich im Rahmen dieser Ausarbeitung aber nur mit der Sicherheit, die man als "Safety" bezeichnet.

Unter **Security** versteht man die Verhinderung von nachteiligen Konsequenzen der beabsichtlichen und unerwünschten Aktionen von anderen. Authorisierter Zugang beispielsweise wird also nicht unterbunden. **Safety** ist die Eigenschaft eines Systemes, dass es unter keinen umständen Menschliches Leben oder allgemein seine Umwelt gefährdet. Im Gegensatz zur Security umfasst dass sowohl beabsichtigte als auch unbeabsichtigte Aktionen von anderen, als auch Unfälle und Fehler in Systemen.

1.1.3 Was ist Sicherheit?

Die Sicherheitsaspekte von Computersystemen[3] lassen sich in drei verschiedene Kategorieren einteilen.

Primäre Sicherheit umfasst die Risiken, die unmittelbar von einem System ausgehen. Bei einer Waschmaschine wären dies beispielsweise die Gefährdung durch Stromschläge, durch auslaufendes Wasser, oder ein durch einen Fehler in der Elektronik verursachter Brand. **Funktionelle Sicherheit** befasst sich mit den Gefährdungen, die von der durch das SkS kontrollierten Anlage ausgehen können. Sie hängt also von der korrekten Funktion der Hardware oder Software des Computersystems ab. **Indirekte Sicherheit** besteht sowohl aus indirekten Auswirkung der Fehler von SkS, als auch aus Mängeln im Mensch-Maschine Interface. Auch nicht funktionale Eigenschaften wie beispielsweise die Bearbeitungsdauer einer Anforderung gehören zu dieser Kategorie.

1.1.4 Formen von SkS

Man unterteilt SkS anhand ihrer Funktion in Kontrollsysteme und Schutzsysteme.

Kontrollsysteme werden benutzt, um die einwandfreie Funktion einer Anlage sicherzustellen. Dies kann beispielsweise ein System sein, was die Temperatur eines Kessel innerhalb eines bestimmten Bereiches regelt, um einen chemischen Prozess aufrecht zu erhalten. Ein Versagen dieses Systems würde eine unmittelbare Gefährdung darstellen, wenn die Anlage gefährliche Stoffe verarbeitet.

[1] Im folgenden wird die Abkürzung "SkS" für den Begriff Sicherheitskritische Systeme verwendet.

[2] Der Begriff "Anlage" kann sowohl eine Fabrik, als auch einzelne, begrenzte Ausrüstungen oder Einrichtungen bezeichnen.

[3] Mit dem Begriff "Computersystem" sind alle Arten von programmierbaren Steuerungen wie Speicherprogrammierbare Steuerungen und Microcontroller als auch zur normale PCs mit Steuerungsaufgaben gemeint.

Das Versagen eines Kontrollsystemes ist also dann sicherheitskritisch, wenn keine alternativen Schutzsysteme eine Abweichung vom sicheren Zustand verhindern können, und eine außer Kontrolle geratene Anlage eine Gefährdung darstellt.

Schutzsysteme benutzen Sensoren um Fehlerbedingungen festzustellen und leiten gegebenfalls Maßnahmen ein, um ihre Auswirkungen zu mindern. Sie werden auch (Not-) Abschaltsysteme genannt.
Eine Notabschaltung in einem Kernkraftwerk wäre ein solches System. Es würde bei erhöhter Temperatur des Reaktorkernes die Kernspaltung durch Einführen von Neutronenabsorbern zum Stillstand bringen.

1.1.5 Unterscheidung Hochintegrierte Systeme - SkS

Als Hochintegrierte Systeme bezeichnet man die Systeme, die verlässlich arbeiten müssen, wobei andernfalls große Verluste oder hoher Schaden auftreten können. Dies schließt die SkS mit ein, umfasst aber beispielsweise auch Kommunikationssatelliten oder Telefon- und Internetvermittlungsknoten. Bei dem Versagen eines Hochintegrierten System muss also nicht zwangsläufig Leben oder die Umwelt gefährdet sein.

1.2 Risiken

Wir sind es in unserem täglichen Leben gewöhnt, die Risiken in unserer Umwelt zu berücksichtigen und ihnen durch geeignete Massnahmen auszuweichen bzw. sie zu verringern. Diese Überlegungen beeinflussen viele grundlegenden Dinge, beispielsweise wo wir wohnen, was für ein Auto wir fahren und was wir essen. Mit Risiken im Bereich der SkS lässt sich leider nicht so intutiv umgehen. Hier brauchen wir einige Definitionen, um Grundlagen für eine Bewertung zu schaffen.

Eine **Gefährdung** stellt eine Möglichkeit für eine potentielle Gefahr dar. Wenn sie in einem Ereignis jemand beeinträchtigt, bezeichnet man sie als **Unfall**. Ein Unfall ist also ein unbeabsichtigtes Ereignis oder eine Kette davon, dass Tod, Verletzungen, Schaden an Umwelt oder Material verursacht. Als **Vorfall** bezeichnen wir ein unbeabsichtigtes Ereignis oder eine Kette davon, dass keine Beeinträchtigung verursacht hat, aber unter anderenUmständen das Potential dazu gehabt hätte.

Die Wichtigkeit einer Gefährdung hängt von den Folgen des Unfalls ab, die es verursachen könnte. Ein Unfall der schwere Konsequenzen hat, ist nur tolerierbar, wenn er fast nie auftritt. Ein Unfall mit minimalen Schaden ist sicherlich häufiger tolerierbar. Das **Risiko** eines Unfalls ist dementsprechend als die Kombination von Frequenz oder Wahrscheinlichkeit seines Auftretens und den Konsequenzen definiert.

1.3 Sicherheitskriterien

1.3.1 Systemanforderungen

Zuverlässigkeit ist die Wahrscheinlichkeit einer Komponente oder eines Systems über eine bestimmte Zeit, innerhalb einer gegebenen Umgebung, zu funktionieren. Dies bedeutet, dass die Zuverlässigkeit über die Zeit abnimmt. Das beispielsweise ein elektrisches Bauteil wie ein Kondensator über eine Woche einwandfrei funktioniert, ist sehr wahrscheinlich. Das er aber über zehn Jahre nicht ausfällt, lässt sich nicht so einfach sicherstellen. Die **Verfügbarkeit** eines Systems ist die Wahrscheinlichkeit, dass es jederzeit funktionieren wird. Diese Größe bleibt damit über der Zeit konstant. Eine Verfügbarkeit von 99 Prozent ist beispielsweise ein Ausfall von 3,65 Tagen im Jahr. Hochverfügbare Systeme haben meist maximal[4] eine Verfügbarkeit von 99,999%, also 5,2 Minuten Ausfall pro Jahr.

[4]Eine höhere Verfügbarkeit ist nur mit unverhältnismässig hohen Aufwand zu erreichen, und wird nur in wirklich sehr kritischen Systemen verwirklicht.

Störungssichere Systeme bieten die erstrebenswerte Möglichkeit, im Falle eines Fehlers in einen sicheren Zustand überzugehen. Bei einer Ampelsteuerung wäre dies beispielsweise der Zustand “Alle Ampeln auf Halt”. Leider lässt sich dies bei manchen SkS nicht erreichen, für ein Flugzeug ist beispielsweise der einzig sichere Zustand am Boden. Als **Systemintegrität** bezeichnet man die Fähigkeit eines Systems, Fehler in seinen eigenen Operationen zu erkennen und mitzuteilen. Dass ist wichtig um gegebenenfalls in einen sicheren Zustand überzugehen, damit fehlerhafte Handlungen oder präsentierte Informationen des SkS nicht zu gefährlichen Zuständen führen. Die **Datenintegrität** ist die Fähigkeit Schaden an seinen Daten zu verhindern, aufgetretene Fehler zu erkennen und wenn möglich zu beheben. Diese Anforderung ist auch bei verschiedenen nicht sicherheitskritischen Systemen von Bedeutung, beispielsweise im Banken oder E-Commerce Bereich.

Zur **Systemwiederherstellung** im Falle eines Fehlers ist es wichtig, ihn zu erkennen, und nach einem Neustart schnell wieder in dem vorherigen Zustand weiterzuarbeiten. Wenn ein System keinen sicheren Zustand besitzt, ist diese Eigenschaft von besonderer Bedeutung. **Wartungsfreundlichkeit** ist die Fähigkeit eines Systems, gewartet zu werden. Als Wartung bezeichnet man die Handlungen, die nötig sind, um ein System in, oder wieder in, die für ihn bestimmten Arbeitsbedingungen zu bringen. Diese Anforderung kann auch quantitativ ausgedrückt werden, dann bezeichnet man sie oft als “mean-time-to-repair”.
Verlässlichkeit ist die Eigenschaft eines Systems, die es rechtfertigt, sein Vertrauen in es zu setzen, und damit von entscheidener Wichtigkeit für SkS. Sie setzt sich zusammen aus den verschiedenen genannten Anforderungen, die je nach System unterschiedlich gewichtet werden.

1.3.2 Konflikte zwischen Systemanforderungen

Bei der Verwirklichung eines SkS muss abgewogen werden, welche der oben aufgeführten Anforderungen priorisiert wird. Es treten unvermeidlich Konflikte zwischen den verschiedenen Systemanforderungen auf. So erlaubt es ein Störungssicheres System, jederzeit in den sicheren Zustand überzugehen. Es wäre dann sehr sicher, seine Verfügbarkeit und Zuverlässigkeit würde aber zu wünschen übrig lassen. Wenn so ein SkS in einer Anlage implementiert würde, wo jeder Stillstand hohe Kosten verursacht, würde sein wirtschaftlicher Erfolg sicherlich ausbleiben. Da die Sicherheit natürlich nicht vollkommen vernachlässigt werden darf, muss ein akzeptabler Kompromiss gefunden werden.

1.3.3 Sicherheitsanforderungen

Der Einsatz eines SkS ist selbstverständlich nur dann sinnvoll, wenn es eine Aufgabe erfüllt. Diese wird in den Sicherheitsanforderungen genau spezifiziert. Um sie zu definieren, muss man die Gefährdungen, die von der Anlage ausgehen, untersuchen, einstufen und entsprechende Methoden festlegen um mit ihnen umzugehen. Weiterhin muss eine Entscheidung über die notwendige Zuverlässigkeit und Verfügbarkeit getroffen, und insgesamt ein angemessenes Integritätsniveau für das SkS gefunden werden. Im Anschluss daran sollte dann die Entwicklungsmethode entsprechend der Vorgaben ausgewählt werden.

Sicherheitsanforderungen schreibt man im allgemeinen als Bedingungen, beispielsweise würde man bei einer Waschmaschine folgendes formulieren: “Erlaube es nicht, die Trommel zu öffnen, solange Wasser in der Maschine ist und der Waschgang noch nicht beendet ist.” Dies wird auch als **Wächtermechanismus** bezeichnet. Mechanismen dieser Art arbeiten meist mit Dingen wie Sperrgittern oder Käfigen.

Ein **Sperrmechanismus** hingegen gibt die gefährliche Funktion immer nur dann frei, wenn sichergestellt ist, dass keine Gefährdungen auftreten können. Sie können nur eingesetzt werden, wenn keine Verzögerungen auftreten, die Anlage also sofort nach der Abschaltung sicher ist. Ein Beispiel für so eine Schaltung wäre eine sogenannte Zweihandsteuerung einer Presse. Hier wird die Funktion nur freigegeben, wenn zwei unabhängige

Schalter gleichzeitig betätigt werden, also keine Hand des Bedieners mehr im Gefahrenbereich ist.

1.4 Fehler

Fehler treten unvermeidlich in allen Systemen auf. Sowohl vollkommende Perfektion im Design als auch absolute Sicherheit gegen zufällige Fehler ist nicht erreichbar. Das Auftreten von Fehlern kann die einwandfreie Funktion von SkS beeinträchtigen. Deshalb sind klare Begrifflichkeiten und Definitionen hier besonders wichtig.

Fehler lassen sich über ihre Eigenschaften in verschiedene Klassen einteilen. Weiterhin muss definiert werden, wie mit aufgetretenen Fehlern umgegangen wird, wie Fehler erkannt und gegebenenfalls korrigiert werden können.

1.4.1 Fehler, Fehlfunktion und Systemausfall

Als **Fehle**r bezeichnet man einen Defekt, eine Fehlerursache im System, der nicht notwendigerweise zu weiteren Problemen führt. Eine **Fehlfunktion** oder Fehlzustand ist eine Abweichung von der benötigten Funktion des Systems. Sie kann aus einem Fehler entstehen. **Systemausfall** tritt auf, wenn das System bei der Durchführung der benötigten Funktionen scheitert. Eine oder mehrere Fehlfunktionen können zu diesem Zustand führen.

1.4.2 Unterscheidung über die Fehlerart

Fehler lassen sich über die Art ihrer Entstehung in zwei unterschiedliche Klassen einteilen.

Zufällige Fehler gehören immer zu Hardware Komponenten von SkS. Da Hardware, auch wenn sie innerhalb ihrer optimalen Arbeitsumgebung eingesetzt wird, immer ausfallen kann, sind alle Systeme anfällig für diesen Fehler. Über statistische Auswertungen und Test lassen sich Aussagen über das Auftreten und die Frequenz von zufälligen Fehlern machen.

Systematische Fehler umfassen beispielsweise alle Software Fehler, die zu den Design Fehlern gehören. Fehler in der Spezifikation gehören auch zu dieser Kategorie. Da sie nicht zufällig auftreten, lassen sich nicht über statistische Auswertungen verhersagen.

1.4.3 Unterscheidung über die Fehlerdauer

Permanente Fehler bleiben unbegrenzt, oder solange bis sie behoben werden, bestehen. Dazu gehören Design Fehler, Fehler in der Spezifikation, wie auch viele Hardware Defekte. Viele Permanente Fehler scheinen periodische Fehler zu sein, weil sie nur zeitweise in Erscheinung treten.

Periodische Fehler treten kurzzeitig, aber wiederholt auf. Kontaktprobleme aufgrund von kalten Lötstellen oder Korrosion an Steckverbindern gehören unter anderem zu dieser Fehlerklasse. Sie sind sehr schwierig zu erkennen und zu beheben, da die Fehlererkennung im Moment des Fehlereintretens passieren muss.

Kurzzeitige Fehler sind beispielsweise die Auswirkung von Höhenstrahlung auf Speichership, oder der Absturz eines Computers aufgrund eines kurzzeitigen Überspannungsimpulses. Obwohl der Fehler nicht mehr auftritt, kann die Fehlfunktion, die er verursacht hat, natürlich weiter bestehen bleiben.

1.4.4 Sonstige Unterscheidungen

Andere Definitionen wie beispielsweise die Unterscheidung zwischen Hardware- und Softwarefehler sind so verbreitet, dass es keiner weiteren Erläuterung bedarf.

Es macht allerdings wenig Sinn, die bekannte Klasse des Menschlichen Fehlers, auch Menschliches Versagen genannt, als eigene Klasse zu definieren. Jedes Versagen eines SkS aufgrund eines Problems während der Entwicklungs-, Implementierungs- oder Wartungsphase stellt letztendlich Menschliches Versagen dar. Dazu gehören insbesondere auch Fehlbedienungen aufgrund einer mangelhaften Mensch-Maschinen Schnittstelle. Auch diese Fehler lassen sich auf eine Schwäche im Entwicklungsprozess zurückführen.

1.5 Fehlervermeidung

Techniken zur Fehlervermeidung zielen darauf ab die Fehler, die während der Designphase von SkS entstehen, zu entfernen. Solche Probleme entstehen hauptsächlich durch Schwächen in der Spezifikation. Da natürliche Sprachen immer ungenau sind, ist es unmöglich, mit ihnen etwas unmissverständlich zu beschreiben. Auch eine korrekte[5] Beschreibung, die für eine sicherere Funktion notwendig wäre, wird durch sie nicht unterstützt.

Digitale Systeme verhalten sich, im Gegensatz zu analogen, nicht stetig. Wegen dieser Eigenschaft, und weil digitale Systeme sich aufgrund der Vielzahl an Kombinationsmöglichkeiten ihrer Eingangsbelegungen nicht vollständig testen lassen, brauchen wir ein anderes Verfahren, um ihr Verhalten zu untersuchen.
Formale Methoden, die auf der diskreten Mathematik und mathematischen Logik basieren, bieten für diese Probleme eine Lösung. Sie erlauben es, die Charakteristiken eines Systems in einer formalen, präzisen Sprache zu beschreiben. Diese kann unmissverständlich interpretiert werden und ihre Korrektheit kann, unter anderem auch durch Hilfsprogramme, bewiesen werden.

Das Anwenden von formalen Methoden während dem vollständigen Entwicklungsprozess ist aber noch sehr aufwendig. Es lohnt sich nur bei SkS mit extrem hohen Integritätsanforderungen. Aber auch ein begrenzter Einsatz kann aber das Vertrauen in die Verlässlichkeit des entwickelten Systems steigern.
Auch die Anwendung von formalen Methoden kann aber nicht sicherstellen, dass die Kunden- oder Sicherheitsanforderungen richtig erfasst worden sind, dass die erstellte Spezifikation korrekt im Sinne ihrer eigentlichen (Sicherheits-) Aufgabe ist.

1.6 Fehlerbehebung

Ausführliches Testen zur Fehlerbehebung, sowohl von Hardware als auch Software, soll sicherstellen, dass möglichst wenig Fehler im fertigen System vor seinem Einsatz vorhanden sind. Testen ist also der Prozess um ein System zu verifizieren oder es zu validieren. Zu diesem Prozess gehört beispielsweise dass direkte Ausführen des Programmes, als auch die Analyse des Quelltextes und des Designs.

Allgemein kann man davon ausgehen, dass je später ein Fehler gefunden wird, desto aufwendiger und teuerer ist seine Behebung. Nicht nur aus diesem Grund ist es wichtig, beim Entwurf von SkS im voraus das Testen einzuplanen und gegebenfalls durch entsprechende Vorkehrungen beim Design zu erleichtern.

Da das Testen aller möglichen Eingangswerte bei digitalen Systemen aufgrund der großen Anzahl von Kombinationen unmöglich ist, ist es wichtig, einen Testplan aufzustellen. Hierbei müssen die notwendige Testabdeckung und die verwendeten Methoden spezifiziert werden, und im Anschluss darauf überprüft werden, ob sie für die benötigte Integrität des Systems ausreichend sind.

[5] Mit “korrekt” sind hier auch die Eigenschaften vollständig und konsistent gemeint.

Ausreichendes Testen, beginnend von der Entwicklungsphase, über die Validierung des Designs bis zur Fertigung des Entwurfes ist sehr wichtig, um ein SkS mit ausreichender Verlässichkeit zu entwickeln.

1.7 Fehlerkennung

Man benutzt Fehlererkennung, um Probleme im fertigen System während dem Betrieb zu erkennen, so dass sie möglichst wenig Auswirkungen auf seine Funktion haben. Zu diesen Techniken gehört beispielsweise das Verwenden von Informationsredundanz, Hardwareüberwachung durch Watchdog timer und eine allgemeine Funktionsprüfung durch eingebaute Prüfroutinen beim Start.

1.8 Fehlertoleranz

Fehlertoleranz soll einen störungsfreien Betrieb trotz dem Auftreten von Fehlern sicherstellen. Die gebräuchlichste Methode, um dies sicherzustellen, ist das verwenden von Redundanz.

Bei der Hardware unterscheidet man zwischen der statischen Rendundanz, wobei mehrere gleiche Hardwarekomponenten über einen Entscheider mit dem Ausgang verbunden werden. Der Entscheider "maskiert" das Ausfallen von $\frac{n-1}{2}$ Modulen, danach beinträchtigt ein Fehler die Funktion des Gesamtverbunds. Im Gegensatz dazu werden Fehler bei der dynamischen Redundanz "kontrolliert". Ein oder mehrere Module des Systems werden in Bereitschaft gehalten, und bei erkennen eines Fehlers gegen das fehlerhafte Modul ausgetauscht. Die hybride Redundanz stellt eine Mischform der beiden genannten Typen dar. Ausserdem erstrebenswert ist es, wenn möglich, einen zusätzlichem nicht programmierbaren Kanal[6] zu benutzen, da ein solches System aufgrund seiner geringeren Komplexität leichter zu beherschen ist.

Redundanz in Software lässt sich zum einen durch mehrere unterschiedliche Implementierungen, die wieder mit einem Entscheider verbunden sind, erreichen. Alternativ kann man eine Implementation mit mehreren redundant ausgelegte Programmblöcken verwenden, wobei nach Ausführung eines Blockes jeweils das Ergebnis überprüft, und im Fehlerfall zum nächsten Block übergegangen wird. Aufgrund des hohen Aufwands und damit der hohen Kosten von Softwareredundanz werden diese Möglichkeiten nur bei sehr kritischen Systemen eingesetzt.

Informationsredundanz wird hauptsächlich in der Kommunikationstechnik eingesetzt. Ohne das Verwenden von Techniken wie Paritätsbits, Checksummen und Fehlererkennende- bzw. korrigierende Codes wäre eine funktionierende Datenübertragung über größere Entfernungen nicht möglich.

Unter zeitliche Redundanz versteht man beispielsweise, dass ein bestimmter Signalpegel über eine definierte Zeit anstehen muss, bevor er registriert wird. Damit unterdrückt man wirkungsvoll kurzzeitige Fehler. Wiederholte Berechnungen mit anschließendem Ergebnisvergleich gehören auch zu dieser Kategorie.

2 Der Stromausfall am 14. August 2003 in den USA

2.1 Einleitung

In einigen Gebieten an der Ostküste der USA, unter anderem in New York, Detroit und in Kanada, wo vor allem Ottawa und Toronto betroffen waren, fiel am 14. August 2003 nachmittags (New Yorker Ortszeit) der Strom weiträumig aus. Acht Staaten im Nordosten der USA und Teile Kanadas mussten bis zu fünf Tage lang ohne elektrische Energie auskommen, dabei waren an die 50 Millionen Menschen betroffen. Insgesamt wurde ein

[6] Beispielsweise ein System bestehend aus einfachen Relais und Sensoren, die fest miteinander verdrahtet sind.

Schaden von schätzungsweise 6 Mrd. US $ verursacht, alleine 1 Mrd. davon in New York [Meldung, Heise], [Schaden, ZDF]. Am Ende der Kettenreaktion, die zu dem Ausfall führte, fehlte eine Leistung von 61 Gigawatt in den Netzen der Stromversorger [Leistung, ZDF]. Wie im offiziellen Untersuchungsbericht [Report, US-Canada Taskforce] berichtet, fielen insgesamt 265 Kraftwerke aus.

2.2 Gründe

Im Untersuchungsbericht werden vier Gründe für den Zusammenbruch des Stromnetzes genannt. Der Zusammenbruch begann in Ohio, in einer Kontrollgebiet dass von der Firma First Energy[7], einem Zusammenschluss von mehreren Stromversorgern, betrieben wurde. Für den einwandfreien und sicheren Betrieb ist der Regionalbezirk ECAR, einem Unterbereich der NERC[8] zuständig. Ausserdem existieren mehrere "Reliability Coordinator", die den einwandfreien Betrieb der einzelnen Kontrollgebiete untereinander sicherstellen. Das betreffenden Gebiet, in dem der Zusammenbruch begann, wurde von "MISO" und "PJM" betreut. Im folgenden werde ich die Gründe, die zur der Katastrophe führten, einzeln aufführen.

2.2.1 Mangelhaftes Systemverständnis und Überlastung des Netzes

Sowohl bei der Überwachungsbehörde ECAR als auch bei FE war das Verständnis für die kritischen Punkte des Netzes von FE nicht gegeben. Der 14. August war ein warmer Sommertag, mit normaler Netzlast[9]. Mehrere Generatoren waren an diesem Tag wegen Wartung außer Betrieb, außerdem waren mehrere große Kondensatorbänke für eine Routineinspektion vom Netz getrennt. Ab Mittags fielen dann mehrere Stromleitungen und ein Kraftwerk, dass viel Blindleistung[10] lieferte, aus. Dies, in Verbindung mit der steigenden Last am Nachmittag, sorgte dafür, dass die Spannung im Netz immer weiter abnahm. Trotz dieser widrigen Umstände und Ausfälle war das Netz von FE bis 15:05[11] stabil, obwohl es in mehreren Parametern an den Grenzen der Vorschriften von NERC operierte.

Diese Ausfälle oder eventuell das warme Wetter sind also nicht für die die Katastrophe verantwortlich. Es war den Verantwortlichen bei FE aber nicht bewusst, dass ihr Netz in diesem Maße ausgelastet war. Die Regionale Kontrollbehörde ECAR hatte es außerdem versäumt, die kritischen Bereiche des FE-Netzes genauer untersuchen, und FE hatte einige vereinbarte Standards ein wenig zu seinen Gunsten gedehnt.

Die gesamten Vorfälle sorgten dafür, dass die Bedienern in den Leitständen sehr wenig Spielraum hatten, um auftretene Probleme zu lösen. Außerdem waren sie nicht ausreichend für Notfälle geschult. Dies verstärkte die Probleme durch mangelhaftes Verständnis und Netzplanung natürlich noch.

2.2.2 Ungenügendes Situationsbewusstsein und technische Probleme bei FE

Ab 14:14 fielen in der Kontrollzentrale von FE nacheinander die Elektronischen Warn- und Logfunktionen, die kritische Netzzustände und Ausfälle wichtiger Komponenten melden, aus. Wenig später fiel dann der Hauptserver aufgrund eines Softwarefehler aus. Der automatisch einspringende Backupserver kam mit der Anzahl der aufgelaufenen Warnmeldungen nicht zurecht, und ging um 14:54 offline. Der Fehler im von FE verwendete Visualisierungs- und Bediensystem "XA/21" [GE Energy, XA/21], der zu dem Ausfall geführt hatte, wurde erst nach wochenlangen Quellcode Untersuchungen gefunden.

[7] Im folgenden abgekürzt als "FE".

[8] North American Electric Reliability Council

[9] Obwohl aufgrund der vielen Klimaanlagen ein hoher Bedarf an kapazitiver Blindleistung war.

[10] Blindleistung ist wichtig, um die Spannungshöhe im Netz zu stabilisieren. Ohne (kapazitive) Blindleistung zur Kompensation bricht die Spannung zusammen.

[11] Alle Zeiten in diesem Kapitel in "EDT", Eastern Daylight Time.

Durch den Ausfall wurden die Netzzustandsdaten nicht mehr aktualisiert, ein Umstand der den Bedienern aber erst sehr viel später auffiel. Auch nach einen Neustart des Server, der die übrigen Fehler beseitigte, blieb das Alarmsystem offline. Die IT-Ingenieure von FE hielten keinerlei Rücksprache mit den Bedienern, sie führten keine Funktionsüberprüfung im Kontrollraum durch.

Das verwendete System war insgesamt nicht auf dem neusten Stand, es gab keinerlei Erfahrung bei den Systemingeneuren mit dieser Art von Fehler. Einigen Bedienern fiel die fehlerhafte Funktion des Systems mehrmals auf, sie versäumten es aber, dies ihren Kollegen mitzuteilen. Auch mehrere telefonische Meldungen über Probleme von benachbarten Kontrollgebieten sorgten nicht dafür, dass die Bediener von einem Fehler in ihrem System ausgingen. Dies wurde dadurch begünstigt, dass die Bediener räumlich getrennt untergebracht waren, und auch kein gemeinsames Logbuch benutzten. Die Bediener waren für Notfälle nicht genügend ausgebildet, es dauerte sehr lange bis sie das Versagen ihres Systems akzeptierten.

2.2.3 Unzureichendes Beschneiden der Bäume

Von 15:05 bis 15:41 fielen drei 345 KV Leitungen im Netz von FE aufgrund von Kontakt mit zu hoch gewachsenen Bäumen aus. Dadurch sank die Spannung im Netz auf knapp 93 Prozent der Nennspannung. Durch die erste ausgefallene Leitung wurde die Belastung der anderen verstärkt, sie wurden mehr erwärmt und sanken dadurch noch mehr durch. Deshalb wurde die Wahrscheinlichkeit für einen Ausfall größer, und Versuche, die Leitungen wieder zu benutzen, schlugen wieder nach kurzer Zeit wegen einem Kurzschluss fehl. Nachdem diese drei Leitungen ausgefallen waren, wurden die verbliebenden 138 KV Leitungen weit über ihre Spezifikation hinaus belastet. Von 15:39 bis 15:58 fielen sieben Leitungen von ihnen aus, alle berührten aufgrund von Überlastung benachbarte Leitungen, Bäume oder den Grund.

2.2.4 Falsche oder keine Echtzeitüberwachungsdaten und mangelhafte Zusammenarbeit

Von 12:15 bis 16:04 war der “State Estimator”, ein Werkzeug zur Entscheidungshilfe, bei MISO nicht funktionsfähig. Ausserdem fehlten Onlinedaten von einer speziellen Stromleitung, aufgelaufene Fehlermeldungen aus dem Netz von FE wurden nicht automatisch mit berücksichtigt. Deshalb war der Reliability Coordinator nicht in der Lage, frühzeitig die Probleme zu erkennen, und FE bei der Lösung zu helfen. Auch nachdem mehrere Leitungen ausgefallen waren, benutzte MISO immer noch die alten Vorhersagewerte des “Flowgate monitoring tool”, einem Werkzeug zur Vorhersage bzw. Berechnung des Stromfluss.

Deshalb fiel es MISO nicht auf, dass ab 15:05 sich das Netz von FE in einem kritischen Zustand befand, die sogenannte $n-1$ Bedingung verletzte. Diese besagt, das ein Netz immer in einem Zustand sein muss, dass es auch nach einen Ausfall des wichtigsten Generators oder Leitung verfügbar bleibt. Wenn ein solcher Zustand eintritt, muss der Netzbetreiber innerhalb einer halben Stunde Maßnahmen ergreifen, um dieser Bedingung wieder zu genügen.

Ausserdem fehlte es bei PJM und MISO an Vorschriften, wie und wann mit Verletzungen von NERC Vorschriften, wie der $n-1$ Bedingung, die an den Grenzen der jeweiligen Zuständigkeitsbereiche auftreten, umgegangen werden soll. Die Reliability Coordinatoren hatten nur ungenügend Einblick in die Struktur des FE Netzes, auch deshalb waren sie sich unsicher, wie die gemeldeten Störungen zu bewerten waren.

2.3 Weiterer Verlauf der Ereignisse

Um 15:59 fielen weitere fünf 138 KV Leitungen aus. Dies verursachte schließlich um 16:05:57 den Ausfall der 345 KV Leitung “Sammy-Star”. Bis zu diesem Zeitpunkt wäre

die Kaskade, die zu dem Blackout führte, immer noch durch automatischen oder manuellen Lastabwurf zu verhindern gewesen. Durch diesen Ausfall aber schalteten sich weitere Kraftwerke und Leitungen ab. Um 16:09:06 fiel dann die Leitung “East Lima-Fostoria” aus. Dies verursachte eine Umkehrung des Energieflusses von Michigan nach Ontario, eine Änderung von etwa 700 MW. Durch diesen abrupten Wechsel der Belastung fing das Stromsystem für mehrere Sekunden an zu schwingen. Dadurch wurde weiter Stress im System verursacht, so dass weitere Stromerzeuger und Leitungen ausfielen.

Um 16:09:38 fielen dann die letzten Leitungen die Michigan und Ohio belieferten aus und eine weitere Energiewelle von ca. 4 GW suchte sich einen neuen Weg nach Michigan. Diese Welle verursachte auf ihrem Weg sehr große Belastungen in den Netzkomponenten. Dadurch fielen innerhalb von einigen Sekunden alle Leitungen und Kraftwerke auf dem Weg des Energieflusses aus, das Netz teilte sich in mehrere Teilbereiche auf. Von 16:10:46 bis 16:12 brach das Netz der Eastern Interconnection aufgrund der Aufteilung schließlich vollständig auseinander. Die einzelnen verbliebenden Inseln konnten die Balance zwischen Energieerzeugung und Verbrauch nicht halten, und automatische Sicherheitsschaltungen schalteten die verbliebenden Kraftwerke und Leitungen ab. Um 16:13 war der Prozess an seinem Ende angelangt und fünzig Millionen Menschen waren ohne Strom. Für eine detailierte Aufstellung sei auf den offiziellen Bericht [Report, US-Canada Taskforce] verwiesen.

2.4 Anmerkungen

An diesem Ereignis lassen viele Kernprobleme kritischer Infrastrukturen aufzeigen. Die hohe Komplexität, sowohl des verwendeten XA/21 Systems, als auch des ganzen Stromnetzes machen es sehr schwierig, es zu beherschen. Weder FE noch die Kontrollbehörde haben die kritischen Punkte des von ihnen versorgten Netzes erkannt. Die hohe Komplexität erfordert die Verwendung von technischen Hilfsmitteln, um überhaupt Entscheidungen treffen zu können. Wenn diese Werkzeuge, wie das verwendete Visualisierungsystem kritische Fehler aufweisen, können dadurch neue, gefährlichere Situationen verursacht werden. Der Fehler im XA/21 System trat nur bei einer extremen Häufung von bestimmten Eingabedaten auf, auf die während der Produktion wohl nicht getestet wurde. Das der Fehler erst nach mehrwöchigen Codeaudit gefunden wurde, weist zusätzlich auf die zu hohe Komplexität und eventuell eine ungenügende[12] Implementation hin.

Der scharfe Wettbewerb und der dadurch entstehende Preisdruck hat sicherlich auch einen großen Anteil an dem Ereignis gehabt. Sicherheit und Verfügbarkeit gibt es nicht umsonst, selbst durch so banale Ursachen wie mangelhaftes zurückschneiden der Bäume können erhebliche Probleme auftreten. An Schulungen, sichereren Systeme, Wartung und einer ausreichenden Netzauslegung sollte man auch nicht sparen, dies hat dieses Ereignis deutlich gezeigt.

Da das verwendete System auf ein handelsübliches Unix Betriebsystem aufsetzt, können auch dadurch neue Gefährungen auftreten. Neue Visualisierungs- und Steuersoftware für Kraftwerke oder Industrieanwendungen verwenden mittlerweile auch Microsoft Windows als Betriebsystem. Kritische Prozessdaten werden teilweise über öffentliche Datennetze wie das Internet befördert.

Wie in [Ohio, Slammer] beschrieben, drang der Windows Wurm “Slammer” Anfang 2003 in ein Kraftwerk in Ohio ein, und deaktivierte für mehrere Stunden ein Sicherheitssystem und die Datennetze. Die Verwendung von handelsüblicher Software und Hardware für Sicherheitskritische Anwendungen ist abzulehnen, da dadurch keine ausreichende Verlässlichkeit sichergestellt werden kann, wie dieses Beispiel zeigt.

Leider wird durch zunehmenden Kostendruck wohl die Verwendung von ungeeigneter Hard- und Software in Sicherheitskritischen Bereichen zunehmen, so dass in Zukunft wohl vermehrt mit Ausfällen von kritischen Infrastrukturen zu rechnen ist.

[12] Ungenügend im Sinne einer schlechten Architektur.

3 Der Flugzeugabsturz bei Überlingen am Bodensee

3.1 Einleitung

“Am 01. Juli 2002 um 23:35:32 Uhr stoßen in etwa 11.000 Meter Höhe über dem nordwestlichen Bodenseeufer eine russische Tupolew-154 der Bashkirian Airlines und eine Boeing-757-Frachtmaschine im rechten Winkel zusammen und stürzen ab. Alle 71 Insassen sterben. Die Tupolew hatte 69 Menschen an Bord, überwiegend Schulkinder aus Ufa, Hauptstadt der russischen Teilrepublik Baschkirien. In der Boeing saßen Pilot und Co-Pilot.“(Auszug aus der [Stuttgarter-Zeitung].)

Technische Mängel bei der Schweizer Flugsicherung und Fehler des Fluglotsen führten zu dem Zusammenstoß. Die Piloten der Tupolev bekamen vom Lotsen und dem bordeigenen TCAS System wiedersprüchliche Informationen, sie treffen die falsche Entscheidung und befolgten die Anweisungen des Fluglotsen. Die Boing flog entsprechend den Anweisungen von TCAS. Die kurz vor der Kollision bei Sichtkontakt eingeleiteten Ausweichmanöver beider Maschinen reichten nicht aus, um die Kollision zu verhindern.

3.2 Das TCAS System

3.2.1 Hintergrund

Mehrere Flugzeugkollisionen in der Luft gaben in den fünfziger Jahren in den USA den Anstoß für die Erforschung von Annäherungswarnsystemen. Eine detailierte Aufstellung der Entwicklung dieser Systeme findet man unter [allstar]. Unterschiedliche Konzepte und Ideen wurden diskutiert, bis im Jahre 1981 die Amerikanische Flugsicherheitsbehörde ein Programm zur Entwicklung des TCAS Systems startete. Ein weiteres Unglück im Jahr 1986 in Kalifornien führte dann letztendlich dazu, dass 1994 ein Gesetz verabschiedet wurde, dass die Nutzung dieses Systems in allen Flugzeugen ab einer bestimmten Sitzplatzanzahl in Amerika vorschreibt. Mittlerweile ist die Nutzung auch in Europa und vielen anderen Ländern Pflicht.

3.2.2 Einstufung

Das “Traffic Alert and Collision Avoidance System” ist ein Annäherungswarnsystem, das einen Zusammenstoß zweier Flugzeuge in der Luft verhindern soll. Es soll dem Piloten unterstützen, kann aber nicht die Kontrolle des Flugverkehrs vom Boden ersetzen. Da es nur ein Warnsystem ist, muss der Pilot angemessen auf die Hinweise und Warnungen des Systems reagieren, es greift nicht in die Flugzeugsteuerung ein [eddh].

Man unterscheidet drei unterschiedliche Versionen des Systems. **TCAS I**, das einfachste (und billigeste) gibt dem Pilot nur eine Anzeige des Verkehrs und eine Navigationsempfehlung[13] aus, die ihn bei der Navigation unterstützen soll. Die Reichweite liegt bei etwa 6 NM[14], ein eventuelles Ausweichen muss entweder über Sicht erfolgen, oder von der Bodenkontrolle unterstützt werden. Es erlaubt dem Piloten, dann um 300 ft zu steigen, seitliche Ausweichbewegungen sind nicht erlaubt. Dieses System ist für alle Flugzeuge mit 10 bis 30 Sitzplätzen in den USA obligatorisch. Zusätzlich zu dieser Funktionalität unterstützt das **TCAS II** System auch vertikale Ausweichempfehlungen[15], bei einer Reichweite von ca. 16 NM. Die Piloten können also, ohne auf die Bodenkontrolle zu warten, selbst schnell ausweichen. Dieses System erlaubt es um 2500 ft zu steigen, oder abzusinken. Die Ausweichempfehlungen sind sowohl vorbeugend, also den Rat die momentane Geschwindigkeit und Höhe beizubehalten, als auch aktiv, die sofortige Aktion erfordern. Es ist für Flugzeuge ab 30 Sitzplätzen in den Vereinigten Staaten Pflicht. **TCAS III** arbeitet über eine

[13] Im Englischen als “Traffic Advisory”, TA bezeichnet.
[14] Seemeile, oder auch nautische Meile. $1NM = 1,852km$
[15] Im Englischen als “Resolution Advisory”, RA bezeichnet.

Reichweite von etwa 20 NM, und berücksichtigt dann auch die Möglichkeit des seitlichen Ausweichens.

3.2.3 Technische Funktionsweise

TCAS ist ein aktives System. Es fragt von Flugzeugen in Reichweite ihre Transponder ab, Luftfahrzeuge ohne Transponder werden nicht erkannt. Durch die Laufzeit des Signals kann die Entfernung, durch die Transponderantwort die Höhe des anderen Flugzeugs festgestellt werden. Etwa jede Sekunde wird so eine Überprüfung des umgebenen Luftraums durchgeführt. Nun schätzt das System die Entwicklung der Situation ab, indem es die Annäherungen der einzelnen georteten Flugzeuge hochrechnet. TCAS überprüft nur die Annäherungsrate, um die zugrundeliegenden Algorithmen möglichst einfach zu halten.

Die Ausgabe von Warnungen wird über zwei Mechanismen festgelegt. Bei schnell bewegenden oder Objekten auf direkten Kollisionskurs wird die Einhaltung des “Tau” Bereichs überprüft. **Tau** ist definiert als die Zeit, die ein Eindringling bis zum nächsten Annäherungspunkt braucht. Damit passt sie sich an die Geschwindigkeit und Richtung des Eindringlings an. Vor einem schnellen Flugzeug auf Kollisionskurs wird also früher gewarnt als vor einem langsamen Objekt auf einem schrägen Kurs. Wenn aber ein Flugobjekt sich langsam annähert, und zusätzlich auch nur einen wenig unterschiedlichen Kurs hat, würde eine Kontrolle von Tau nicht mehr ausreichen, um auszuweichen. Deshalb wird zusätzlich der **Mindestabstand** für Objekte überprüft. Wenn ein Objekt diese Schwelle unterschreitet, wird auch eine Warnung ausgegeben. Wenn keine genauen Höhenangaben von dem eindringenden Flugobjekt verfügbar sind, wird ein fester Wert angenommen. Reelle Werte für diese Paramter findet man unter [aerowinx].

Diese Systemparameter werden abhängig von der Flughöhe angepasst. Direkt nach dem Start oder bei der Landung werden beispielsweise keine Ausweichempfehlungen ausgegeben. In dieser Phase ist zum einen die Wahrscheinlichkeit für Fehlalarme sehr hoch, ausserdem gibt es meist keine Möglichkeit, Ausweichbewegungen vorzunehmen. Ausserdem wird durch die Towerkontrolle des Flughafens sichergestellt, dass keine Kollisionen auftreten können.

Wenn Ausweichempfehlungen gegeben werden, wird dem TCAS System des Eindringlings[16] mitgeteilt, ob ein Steigen oder Fallen des Flugzeuges angeordet worden ist. Damit wird sichergestellt, dass beide Flugzeuge auf entgegengesetzen Ausweichkurs gehen.

3.2.4 Mensch - Maschine Interface

Der allgemeine Überblick über die Verkehrssituation wird meist auf dem sogenannten primären Display gegeben. Es bietet eine graphische Darstellung des eigenen Flugzeuges und aller anderen Flugobjekte in Reichweite. Relative Höhe und Distanz werden angezeigt, wenn möglich auch der Kurs des Eindringlings. Eine Navigationsempfehlung wird am Display dargestellt, und zusätzlich einmal über eine akustische Meldung signalisiert. Bei einer Ausweichempfehlung wird zusätzlich zur normalen Anzeige auf dem Display mehrmals die richtige Handlung angesagt, also beispielsweise “Steigen, steigen” bei Annäherung eines Flugzeugs von unten.

3.2.5 Probleme

Zur Einführung des TCAS Systems wurde es von den Piloten nur als Ärgernis betrachtet. Die große Anzahl von Fehlalarmen erzeugte nur zusätzlichen Stress. Die erste Version der internen Software war viel zu empfindlich und zeigte einige interessante Probleme. So wurden beispielsweise Transponder auf Brücken und Schiffe als Alarme gemeldet, oder es wurde vor dem eigenen Flugzeug gewarnt. Da Testflüge sehr teuer sind, und ein Testen unter reellen Bedingungen sehr schwierig zu simulieren war, wurden die Logik letztlich

[16] Als Eindringling wird allgemein das jeweils “andere” Flugzeug bezeichnet.

mittels aufgezeichneter Radardaten verfeinert [caasd]. Nach mehreren Revisionen wurde dann aber eine hohe Reduktion der Fehlalarme erreicht. Mittlerweile sehen die Piloten das System als verlässlich an, unter anderen ist die Befolgung von Ausweichempfehlungen bei allen großen Fluggesellschaften für sie Pflicht.

Durch die große Reichweite von TCAS, neuere Versionen erreichen bis zu 60 NM, werden ständig Objekte angezeigt, die überhaupt keine Bedrohung darstellen. Dadurch wird die Aufmerksamkeitsschwelle des Pilotens für das System gesenkt. Die hohen Geschwindigkeiten in der Luftfahrt erlauben es aber nicht, die Reichweite zu senken, da ansonsten die Zeit für eine Reaktion nicht mehr ausreichen würde.

3.3 Gründe

Die BFU[17] hat sich eingehend mit dem Unfallverlauf befasst. In ihrem vorläufigen Untersuchungsbericht [Untersuchungsbericht BFU] stellt sie den Verlauf des Unglücks detailiert dar. Es stellte sich heraus, dass eine Verkettung unterschiedlicher, teilweise unglücklicher Umstände zu dem Unglück führte.

3.3.1 Technische Mängel bei der Flugsicherung

Zum Zeitpunkt des Unglücks war die Schweizer Flugsicherung “Skyguide” nicht voll einsatzfähig. Aufgrund von Wartungsarbeiten war das Bodenkollisionswarnsystem STCA nicht funktionsfähig. Dadurch bekam der Lotse keine frühzeitige Kollisionswarnung, sondern musste die Flugbahnen visuell auf Kollisionen überwachen. Kurz vor der Katastrophe versuchte eine benachbarte Bodenkontrolle den Lotsen telefonisch zu erreichen, aufgrund von Wartungsarbeiten an der Telefonanlage und Störungen an der Ersatzleitung kam aber keine Verbindung zustande. Dem Lotsen war der Ausfall der Warnsysteme nicht bewusst, dadurch musste er falsche Annahmen über den von ihm betreuten Luftraum machen. Aufgrund der ausgefallenen Telefonleitung konnten auch benachbarte Flugsicherungen keine Warnmeldung an ihn weitergeben.

3.3.2 Stress und Überlastung

Zum Zeitpunkt des Unglücks war der Fluglotse alleine, er musste einen Kollegen der gerade Pause machte vertreten. Zusätzlich zu der erhöhten Arbeitsbelastung und dem Ausfall der technischen Überwachungseinrichtung hatte er also die doppelte Arbeitsbelastung zu bewältigen. Dieser erhöhte Druck begünstigte sicherlich die Verwechslung von zwei Flugzeugen auf dem Radarschirm, aufgrund er die falschen Anweisung gab [Verwechslung, Tagesschau]. Durch den Ausfall des Telefons konnte er ein Flugzeug nicht an eine benachbarte Flugsicherung übergeben, auch dadurch wurde seine Belastung erhöht [Übergabe, Tagesschau].

3.3.3 Probleme durch TCAS

Durch die anfänglichen Probleme wurde das Vertrauen der Piloten in die Verlässlichkeit des Systems untergraben. Obwohl bei der Entwicklung von TCAS formale Methoden angewendet wurden [Storey], konnten die Probleme des Systems in der Realität dadurch nicht erkannt werden. Ein ausreichendes Testen mit reellen Daten, wie später durchgeführt, wäre sehr sinnvoll für eine schnelle Akzeptanz von TCAS gewesen. Dann hätten die Piloten der Tupolev vielleicht eher ihrem Warnsystem vertraut, als der Flugsicherung.

3.3.4 Formelle Probleme

Es ist in den Internationalen Bestimmungen zum Flugverkehr unzureichend spezifiziert gewesen, wie sich Piloten im Falle eines Konflikt von Bodenkontrolle und TCAS System

[17] Bundesstelle für Flugunfalluntersuchung

verhalten sollen. Eine direkte Anweisung an deutlich sichtbarer Stelle fehlte, man konnte die vorhandenen Bestimmungen unterschiedlich auslegen. Da von Ausgabe der Ausweichempfehlung bis zur Kollision weniger als 40 Sekunden vergehen, kann dem Piloten eine genau durchdachte Entscheidung nicht zugemutet werden. [Empfehlung BFU]

3.4 Anmerkungen und Konsequenzen

Die Optimierungen an der Software des TCAS Systems sind nahezu abgeschlossen. In der als “final” bezeichneten Version, die auch in Europa vorgeschrieben ist, ist noch einmal eine Reduktion der Fehlalarme erreicht worden, unter anderem auch durch großangelegte Tests mit reellen Radardaten. Eine Studie des Center for Advanced Aviation System Development kam 1994 zu dem Schluss, dass die Verwendung von TCAS die Wahrscheinlichkeit einer beinahen Kollision um 90 bis 98 Prozent verringert [caasd]. Die BFU hat als Reaktion auf diese Katastrophe eine Empfehlung herausgegeben, dass in Zukunft die Piloten bei wiedersprüchlichen Anweisungen von TCAS und der Bodenkontrolle den Anweisungen des TCAS Systems folgen sollen [Empfehlung BFU]. Dies soll international durchgesetzt werden, ausserdem erhalten die Piloten ähnliche Anweisungen in den entsprechende Sicherheitsvorschriften der Fluggesellschaften. Weitere Aufschlüsse zum Unglück könnte der offizielle Untersuchungsbericht geben, der Mitte 2004 herausgeben werden soll [Verzögerung BFU]. Bei der Schweizer Flugsicherung ist die Anzahl der Fluglotsen in der Nachtschicht erhöht worden, auch stehen jetzt Notfall-Handys zur Verfügung.
[Skyguide, DasErste] Dadurch kann hoffentlich in Zukunft ein ähnliches Unglück vermieden werden.

Literatur

[Storey]	Neil Storey, Safety-Critical Computer Systems, Pearson-Prentice Hall, 1996
[Giese]	Giese, Universität Paderborn, Vorlesungs-folien zu SkS, SCCS-I-66-88-2x.pdf, http://wwwcs.upb.de/cs/hg/index_dt.html
[Schneier]	Bruce Schneier, Beyond Fear, Copernicus Books, 2003
[allstar]	http://www.allstar.fiu.edu/aero/TCAS.htm
[eddh]	http://www.eddh.de/topics/tcas.html
[aerowinx]	http://www.aerowinx.de/html/tcas.html
[caasd]	http://www.caasd.org/proj/tcas/
[Stuttgarter-Zeitung]	http://www.stuttgarter-zeitung.de/stz/page/detail.php/235807
[Untersuchungsbericht BFU]	http://www.bfu-web.de/flusiinfo/V163_jb2002.pdf
[Empfehlung BFU]	http://www.bfu-web.de/flusiinfo/V163_jb2002.pdf, Seite 25
[Verzögerung BFU]	http://www.flugzeug-absturz.de/newsarchiv.php?jahr=2004&monat=03&id=1054
[Verwechslung, Tagesschau]	http://www.tagesschau.de/aktuell/meldungen/0,1185,OID989014,00.html
[Übergabe, Tagesschau]	http://www.tagesschau.de/aktuell/meldungen/0,1185,OID891722_TYP1_NAVSPM2~989868_REF,00.html
[Skyguide, DasErste]	http://www.daserste.de/wwiewissen/thema_dyn~id,3htua724orhvp52v~cm.asp
[Meldung, Heise]	http://www.heise.de/newsticker/meldung/39436
[Schaden, ZDF]	http://www.heute.t-online.de/ZDFheute/artikel/1/0,1367,WIRT-0-2060897,00.html
[Leistung, ZDF]	http://www.heute.t-online.de/ZDFheute/artikel/7/0,1367,MAG-0-2060103,00.html
[Report, US-Canada Taskforce]	https://reports.energy.gov/BlackoutFinal-Web.pdf
[GE Energy, XA/21]	http://www.gepower.com/prod_serv/products/scada_software/en/xa21.htm
[Ohio, Slammer]	http://www.securityfocus.com/news/6767

Qualitätsmanagement & Zertifizierung

bei sicherheitskritischen Systemen

Seminararbeit
im Fach Informatik
im Rahmen des Seminars „Sicherheitskritische Systeme"
an der

Universität Siegen, Fachgruppe für Praktische Informatik

eingereicht bei
Diplom-Informatiker Dr. Jörg Niere

vorgelegt von
Christian Dörner

Sommersemester 2004
Siegen, April 2004

Zusammenfassung

Das Qualitätsmanagement und die Zertifizierung spielen eine wichtige Rolle im Zusammenhang mit sicherheitskritischen Systemen. Sie sind Grundbausteine für die Entwicklung und Implementierung eines sicherheitskritischen Systems. Qualitätsmanagement und Zertifizierung sind eng miteinander verknüpft, wobei das Qualitätsmanagement sozusagen die Vorstufe der Zertifizierung darstellt. Beide Vorgänge laufen parallel zur Entwicklung und Produktion ab und es existiert eine Vielzahl an Standards und Normen. Im Falle des Qualitätsmanagements sorgen sie für die korrekte Umsetzung der Qualitätssicherung und Qualitätskontrolle. Bei der Zertifizierung beschreiben sie hingegen Anforderungen an das System, die erfüllt werden müssen, um das angestrebte Zertifikat zu erhalten. Der so genannte *Safety Case* ist dabei ein wichtiger Bestandteil zu Demonstration der Sicherheit des entwickelten Systems.

Inhaltsverzeichnis

1 Qualitätsmanagement **1**
1.1 Qualität . . . 1
1.2 Einleitung . . . 1
1.2.1 Definition: Sicherheit . . . 2
1.2.2 Sicherheitskritische Systeme . . . 2
1.2.3 Hoch-integrierte Systeme . . . 3
1.2.4 Vertrauenswürdige Systeme . . . 3
1.3 Qualitätssicherung . . . 4
1.3.1 Einleitung . . . 4
1.3.2 Wie wird die Qualitätssicherung angewendet . . . 4
1.4 Qualitätskontrolle . . . 5
1.4.1 Einleitung . . . 5
1.4.2 Wie wird die Qualitätskontrolle durchgeführt . . . 6
1.5 Qualitätsstandards . . . 6
1.5.1 ISO 9000 . . . 6
1.5.2 ISO/IEC 9126 . . . 7
1.5.3 Sonstige Standards . . . 7

2 Zertifizierung **8**
2.1 Einleitung . . . 8
2.2 Arten der Zertifizierung . . . 9
2.2.1 Organisationen oder Individuen . . . 9
2.2.2 Werkzeuge oder Methoden . . . 9
2.2.3 Systeme oder Produkte . . . 10
2.3 Ziele der Zertifizierung . . . 10
2.4 Safety Case . . . 10
2.4.1 Einleitung . . . 10
2.4.2 Zweck des Safety Case . . . 11
2.4.3 Probleme bei der Entwicklung . . . 11
2.5 Der Prozess der System-Zertifizierung . . . 11
2.6 Zertifizierungsstandards . . . 12
2.6.1 IEC 61508 . . . 12
2.6.2 DO-178B . . . 13
2.6.3 Sonstige Standards . . . 14

3 Schlussfolgerung **14**

Abbildungsverzeichnis

1 Bestandteile des Qualitätsmanagements 2
2 Expertengruppen bei der Entwicklung des Safety Case 12

1 Qualitätsmanagement

1.1 Qualität

Bevor es möglich ist sich mit *Qualitätsmanagement* auseinander zu setzen, muss zunächst einmal der Terminus *Qualität* näher betrachtet werden. In der Literatur kursieren mehrere Definitionen, die den Begriff näher beschreiben. So wird Qualität zum Beispiel mit den folgenden Eigenschaften beschrieben:

- Tauglichkeit der Nutzung
- Übereinstimmung mit den Anforderungen oder Bedingungen
- Alle Merkmale und Charakteristika eines Produktes oder einer Dienstleistung, die sicherstellen, dass der zuvor bestimmte Bedarf befriedigt wird.

Im Hinblick auf *sicherheitskritische Systeme* und deren gehobenen Anspruch auf Qualität sollte man den Qualitätsbegriff als eine Kombination aus diesen Eigenschaften ansehen. So könnte man Qualität als einen Prozess beschreiben, der dafür sorgt, dass die gefertigten Produkte ihren Zweck mit einer möglichst geringen Zahl an Mängeln erfüllen.
Nachfolgend soll dieser Qualitätsbegriff als Grundlage dienen.

1.2 Einleitung

Was verstehen wir also unter Qualitätsmanagement?
Das Qualitätsmanagement ist als Oberbegriff zu verstehen, der eine möglichst gute Mischung aus *Qualitätssicherung* und *Qualitätskontrolle* vereint (vlg. Abbildung 1), um die besten Ergebnisse im Form eines qualitativen Produktes zu erzielen.

Diese beiden essentiellen Bestandteile des Qualitätsmanagements werden in den Abschnitten 1.3 und 1.4 noch genauer beschrieben. Das Qualitätsmanagement wird im Unternehmen normalerweise mittels eines so genannten *Qualitätsmanagement Systems* umgesetzt.
Die *International Organization for Standardization*, kurz ISO[1], definiert in ihrem Standard ISO 9000[2] ein Qualitätsmanagement System folgendermaßen:

Definition 1 (Qualitätsmanagement System) *The organizational structure, responsibilities, procedures, processes and resources for implementing quality management. [2, S. 348]*

Bevor es möglich ist näher auf Qualitätssicherung und Qualitätskontrolle einzugehen, sind noch ein paar weitere Definitionen notwendig.

[1] Für weitere Informationen siehe [1]

[2] Anmerkung: Mehr als 500.000 Organisationen in 160 Ländern wenden den ISO 9000 Standard an [5]. Nähere Informationen zu ISO 9000 finden sich in Abschnitt 1.5.1.

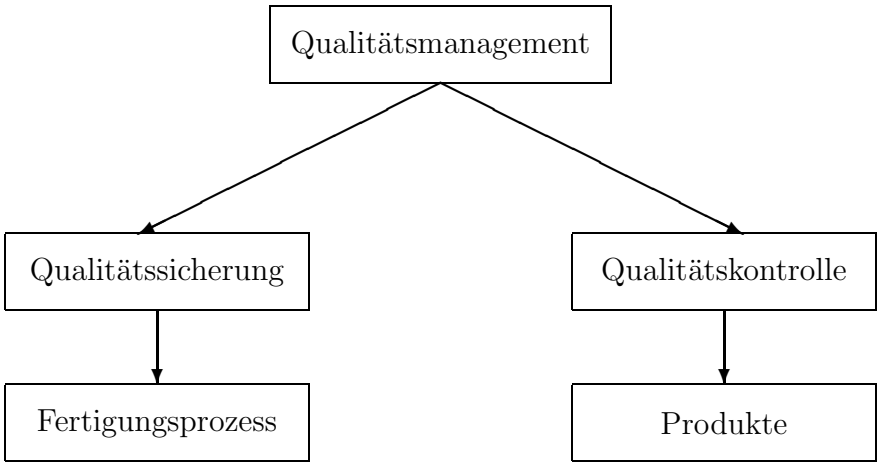


Abbildung 1: Bestandteile des Qualitätsmanagements

1.2.1 Definition: Sicherheit

Ein weiterer wichtiger Terminus, der in diesem Artikel immer wieder auftaucht, ist der Begriff *Sicherheit*. Dieses Schlagwort muss allerdings genauer abgegrenzt werden, da es einen relativ großen Spielraum der Interpretation offen lässt.
Unter Sicherheit wird im Zusammenhang mit sicherheitskritischen Systemen nur das Risiko für das *menschliche Leben* oder der *Umgebung* angesehen, nicht aber die Sicherheit der Privatsphäre oder der nationalen Sicherheit. Sicherheit ist dabei als eine *Systemeigenschaft* anzusehen, daher haben alle Systembestandteile (Hardware, Software und Benutzer) ihren Einfluss darauf.
Nach dieser einleitenden Definition von Sicherheit ist es nun möglich, sicherheitskritische Systeme einzuordnen und abzugrenzen.

1.2.2 Sicherheitskritische Systeme

Ein sicherheitskritisches System lässt sich als ein System charakterisieren, bei dessen inkorrekter Funktion (Versagen) es zu ernsthaften Konsequenzen (Tötung von Personen, schweren Verletzungen, schwerwiegenden Umweltschäden, oder erheblichen Geldbußen) kommen kann.
Diese etwas allgemeiner gehaltene Beschreibung lässt allerdings noch einen gewissen Freiraum übrig. So könnte man vier Unterarten von sicherheitskritischen Systemen bilden, die zu unterschiedlichen Konsequenzen führen:

- Lebens-kritische Systeme
- Missions-kritische Systeme
- Umwelt-kritische Systeme
- Kosten-kritische Systeme

Im Zusammenhang mit Qualitätsmanagement und Zertifizierung sind die *exakten Konsequenzen* eines Systems allerdings nicht von so großer Bedeutung, daher reicht der Begriff sicherheitskritisch vollkommen aus.
Diese Definition grenzt zugleich die sicherheitskritischen Systeme weit genug ein, denn Folgen wie unbefugte *Einsicht*, *Modifizierung* oder *Löschung* von Daten sind nicht unter den oben aufgeführten Punkten zu finden. Somit wird die Deckungsgleichheit mit dem zuvor geprägten Sicherheitsbegriff gewährleistet.

Bei sicherheitskritischen Systemen handelt es sich zumeist um *real-time control systems*[3]. Beispiele für solche Systeme sind:

- Herz-Lungen Maschinen
- Atomreaktoren
- Airbags
- Flugkontrollsysteme

1.2.3 Hoch-integrierte Systeme

Hoch-integrierte Systeme bilden sozusagen die „Obermenge" der sicherheitskritischen Systeme. In dieser enthalten sind auch andere kritische Systeme, wie z. B. Sicherheitssysteme bei Banken.
Einem hoch-integrierten System muss also zugetraut werden, dass es zuverlässig arbeitet, da es sonst zu einem inakzeptablen Verlust oder Schaden kommen kann.

1.2.4 Vertrauenswürdige Systeme

Die *Zuverlässigkeit* eines Computersystems ist die Eigenschaft, die ein gerechtfertigtes Vertrauen des Systems schafft. Genau diese Eigenschaft liegt den vertrauenswürdigen Systemen zu Grunde.
Zwei elementare Aspekte solcher Systeme sind erstens das Vertrauen, dass das System so funktioniert, wie es seine Spezifikationen beschreiben, und zweitens das Vertrauen, dass Gefahren vom System vermieden werden. Definiert man jetzt noch Vertrauenswürdigkeit als eine Kombination aus *Funktionsfähigkeit*, *Verfügbarkeit*, *Sicherheit* und *Vertraulichkeit*, wäre es ohne weiteres möglich, vertrauenswürdige Systeme als *abstrakten Oberbegriff* aller bisher angesprochenen Systeme zu sehen. Von einer solchen Abstraktion sollte allerdings Abstand genommen werden, da diese nur zu Verständnisproblemen aufgrund ihrer zu breit gefächerten Definition führt.

[3]Siehe [3] für nähere Informationen

1.3 Qualitätssicherung

1.3.1 Einleitung

Nach diesen einleitenden Begriffsabgrenzungen ist es nun möglich, wieder zum eigentlichen Thema zurückzukehren.
ISO 9000 definiert Qualitätssicherung wie folgt:

Definition 2 *All those planned and systematic actions necessary to provide adequate confidence that a product or service will satisfy given requirements for quality. [2, S. 348]*

Wie Abbildung 1 bereits deutlich gemacht hat, beschäftigt sich die Qualitätssicherung mit dem *Fertigungsprozess* von Produkten. Das heißt die Qualitätssicherung muss dafür sorgen, dass die Entwicklung, bzw. Design- und Test-Operationen richtig ausgeführt werden. Sie erstreckt sich somit auf diese drei Aufgabenbereiche:

1. Definition des Fertigungsprozesses und des eingesetzten Management Systems
2. Management der Ressourcen
3. Prüfende und qualitätsverbessernde Aktionen

Bei sicherheitskritischen Systemen stehen neben der Verbesserung der Qualität des Produktes auch die Verbesserung der Entwicklung und Produktionseffizienz, sowie die Bereitstellung einer Basis für die Rechtfertigung der Sicherheit (*Safety Case, vgl. Abschnitt 2.4*) im zentralen Blickfeld.
Da Sicherheit nicht alleine durch *Testen* demonstriert werden kann, kommt auch hier der Qualitätssicherung eine besondere Bedeutung zu. Sie muss aufgrund der eingesetzten Methoden bei Entwicklung und Fertigung zur *Akzeptanz* des Systems beitragen.

1.3.2 Wie wird die Qualitätssicherung angewendet

Wie bereits erwähnt, ist die Definition der notwendigen Verfahren und des dazu gehörigen Management Systems ein grundlegender Bestandteil der Qualitätssicherung. Weiterhin muss bei der Umsetzung eine klare Zuordnung von *Aufgaben* und *Personen* stattfinden, um eine klare Aufgabenverteilung zu gewährleisten.

Im Falle von sicherheitskritischen Systemen werden die Management Strukturen normalerweise mit Hilfe von halb-formalisierten Methoden beschrieben, um eine möglichst unmissverständliche Beschreibung zu erhalten.
Bei der letztendlichen Umsetzung der Qualitätssicherung haben diese drei Punkte eine zentrale Bedeutung:

1. **Ressourcen:** Die benötigten *Fachkenntnisse* für jedes Mitglied des Teams werden ermittelt, um sicherzustellen, dass diese menschlichen Ressourcen verfügbar sind.

2. **Prüfung:** Es wird geprüft, ob die Methoden richtig angewendet werden und ob diese erfolgreich sind. Dies wird sichergestellt durch das Sammeln von Daten und deren anschließende Diskussion. Dazu bedarf es unter Anderem einer Prüfung von Proben der benutzten Dokumente.

3. **Qualitätsplan:** Dieser gibt eine detaillierte Auskunft über die Maßnahmen der Qualitätssicherung des konkreten Projektes.

Der Prozess der Qualitätssicherung ist ein notwendiger, aber *nicht ausreichender Bestandteil*, um die Qualität des Endproduktes zu garantieren. Dazu bedarf es auch der Qualitätskontrolle, mit der sich der nachfolgende Abschnitt beschäftigt.

1.4 Qualitätskontrolle

1.4.1 Einleitung

Zu Beginn sei auch hier die Definition von Qualitätskontrolle nach ISO 9000 dargestellt:

Definition 3 *The operational techniques and activities that are used to fulfil requirements for quality. [2, S. 351]*

Das Ziel der Qualitätskontrolle ist es, dafür zu sorgen, dass möglichst keine fehlerhaften Produkte das Unternehmen verlassen. Das heißt die Qualitätskontrolle muss sicher stellen, dass das Endprodukt die zuvor festgelegten *Spezifikationen* erfüllt. Sollte dies nicht der Fall sein, so muss sie für eine entsprechende Anpassung des Produktionsprozesses sorgen.
Bei *Hardwarekomponenten* muss nicht nur deren Funktionalität überprüft werden, sondern auch komplexere Charakteristika wie die *Zuverlässigkeit*, *Haltbarkeit* und *Sicherheit*. Unglücklicherweise sind solche Charakteristika nicht allzu leicht zu messen, da es hier nicht immer Standardgrößen als Maßstab gibt, und das Messen an sich auch Probleme bereitet.
Bei *Software* sind die Probleme noch exorbitanter. Erstens ist es generell unmöglich ein Programm komplett zu testen, um seine Übereinstimmung mit den funktionalen Aspekten der Spezifikation zu determinieren. Zweitens beinhalten die Anforderungen an Software Eigenschaften wie *Verlässlichkeit*, *Effizienz* oder *Portierbarkeit*[4]. Hier tritt auch wieder das Problem auf, dass diese extrem schwer zu messen sind.

[4] Übertragbarkeit von einer Hardware-Plattform auf eine andere

Schlussfolgern kann man daher, dass Qualitätskontrolle von der Fähigkeit des Messens der Parameter des Produktes abhängt, das wir kontrollieren möchten.

1.4.2 Wie wird die Qualitätskontrolle durchgeführt

Viele Unternehmen benutzen statistische Prozesse, um die Qualitätskontrolle durchzuführen und deren Fehler systematisch zu reduzieren. Dabei werden zufällig Exemplare des *Output* ausgewählt, ausprobiert und getestet. Die dabei festgestellten *Toleranzen* zwischen den Spezifikationen und dem Produkt werden zur Anpassung des Produktionsprozesses genutzt, um in Zukunft keine Produkte minderer Qualität zu produzieren.

Eine recht populäre Methode in diesem Zusammenhang sind die so genannten „sechs Sigma Stufen der Qualität[5]". Ein Unternehmen, das diese „Qualitätsstufe" erreicht, schafft es, seine Fehlerwahrscheinlichkeit auf unter vier Teile pro einer Million produzierter Teile zu senken.

1.5 Qualitätsstandards

Aus den vorherigen Abschnitten geht hervor, dass Standards (z. B. ISO 9000) existieren, die bei der Herstellung von qualitativ hochwertigen Produkten mithelfen.
Qualitätsstandards existieren seit über 150 Jahren. Zu Beginn wurden sie vor allem im *Eisenbahnsektor* verwendet. Hier seien nun zwei der wichtigsten Qualitätsstandards kurz vorgestellt.

1.5.1 ISO 9000

Der Standard geht ursprüngliche aus dem BS 5750 Standard des *British Standard Institute* hervor. Heutzutage wird er unter dem Namen ISO 9000 von der *International Organization for Standardization* definiert. In Europa ist der Standard auch unter der Bezeichnung *EN 2900x* bekannt.
Unternehmen, die ihr Qualitätsmanagement System nach diesem Standard ausrichten, können sich dies entsprechend ISO 9000 zertifizieren lassen. Der Standard ist einer der weltweit wichtigsten industriellen Qualitätsstandards. Mehr als *500.000 Organisationen* in *160 Ländern* wenden den ISO 9000 Standard an [5]. Die neueste Revision ISO 9000:2000 datiert auf das Jahr 2000.

ISO 9000 spezifiziert die Anforderungen eines Qualitätsmanagement Systems, und beschäftigt sich mit der Qualitätssicherung und Qualitätskontrolle. Er stellt

[5]Siehe [4] für weitere Informationen

nicht die Qualität eines Produktes sicher, sondern bescheinigt vor allem, dass der Ablauf des Produktionsprozesses richtig durchgeführt und überprüft wurde.

1.5.2 ISO/IEC 9126

Der ISO/IEC[6] 9126 Standard bezieht sich primär auf die Qualitätskontrolle bei *Computer Software*. Der Standard definiert sechs Charakteristika, die die Qualität von Software determinieren: [2, S. 354]

1. **Funktionalität:** Attribute, die dafür sorgen, dass die spezifizierten Eigenschaften eingehalten werden. Funktionen sind nur solche, die implizierte oder festgelegte Bedürfnisse befriedigen.

2. **Zuverlässigkeit:** Attribute, die zeigen, dass die Leistungsfähigkeit von Software auf ihrem Anwendungsniveau, unter festgelegten Bedingungen, innerhalb einer festgelegten Zeit, aufrecht erhalten werden kann.

3. **Brauchbarkeit:** Attribute, die den Aufwand der Anwendung sowie die individuelle Auswertung eines solchen Nutzens unter festgelegten Bedingungen beschreiben.

4. **Effizienz:** Attribute, die die Leistungsfähigkeit von Software und deren benutzte Ressourcen unter festgelegten Bedingungen vergleichen.

5. **Haltbarkeit:** Attribute, die den Aufwand einer durchzuführenden Anpassung beschreiben.

6. **Portierbarkeit:** Attribute, die die Übertragbarkeit von Software von einer Plattform auf eine andere Plattform beschreiben.

1.5.3 Sonstige Standards

Nach der Ausführung dieser beiden wichtigen Standards seien hier noch zur Komplettierung ein paar weitere Standards aus dem zivilen und militärischen Sektor genannt:

- Zivile Standards:
  - ANSI/IEEE Standard 730-1989 *Software Quality Assurance Plans* [6]
  - ASME Standard NQA 2a *Quality Assurance Requirements for Nuclear Facility Applications* [7]
- Militärische Standards:

[6] International Electrotechnical Commission (Genf, Schweiz)

- Department of Defence, Standard-2168 *Defense System Software Quality Program* [8]
- Ministry of Defence, Defence Standard 00-16 *Guide to the Achievement of Quality in Software* [9]
- NATO, Quality Standard AQAP-150 *Requirements for Quality Management of Software Development* [10]

2 Zertifizierung

2.1 Einleitung

Qualitätssicherung ist auf jeden Fall ein wichtiger Bestandteil, um die Qualität eines Produktes sicherzustellen. In manchen Industrien ist die Qualitätssicherung allerdings *nicht ausreichend*, um ein Produkt an den Markt zu bringen. So ist es in der Luftfahrtindustrie zum Beispiel nur dann erlaubt, ein Flugzeug im Luftverkehr einzusetzen, wenn dieses zuvor ein *Lufttüchtigkeitszertifikat* erhalten hat.
In anderen Industrien sind *Zertifikate* nicht unbedingt notwendig, doch es ist eindeutig erkennbar, dass zertifizierte Produkte klar erkennbare Vorteile beim Marketing besitzen.

Doch was bedeutet Zertifizierung genau?
Als Zertifizierung bezeichnet man den Prozess der Ausgabe einer geschriebenen Versicherung (Das Zertifikat) durch eine unabhängige externe Organisation, um die Übereinstimmung mit einem Standard, einem Satz von Richtlinen oder eines ähnlichen Dokumentes nachzuweisen.

Die Zertifizierung wird meist von *staatlichen Behörden* oder von *national angesehenen Organisationen* durchgeführt, um eine möglichst große Vertrauensbasis zu schaffen. In manchen Industrien gibt es Regulierungsbehörden, die die Produktion überwachen und für die Ausgabe von Zertifikaten verantwortlich sind. Hervorzuheben sind hier die Luftfahrt- und Atomindustrie.

Um ein Zertifikat zu erhalten muss der Entwickler in der Lage sein zu zeigen, dass alle wichtigen *Gefahren* identifiziert wurden, und entsprechende Maßnahmen zu deren *Vermeidung* getroffen wurden. Des Weiteren muss er zeigen, dass die Integrität des Systems für dessen Anwendung angemessen ist.
In manchen Fällen muss der *Beweis* erbracht werden, dass das System einem bestimmten Standard entspricht. Außerdem müssen Nachweise über die eingesetzten Entwicklungs- und Testmethoden erbracht werden, um die Angemessenheit ihres Designs zu überprüfen.

Zusätzlich ist es notwendig, erschwerende Argumente anzuführen, um dem Anspruch der *Systemsicherheit* gerecht zu werden. Um diese Beweise zu erbringen, wird im allgemeinen der so genannte *Safety Case* (siehe Abschnitt 2.4) entwickelt.

Abschließend bleibt noch anzumerken, dass die Zertifizierung eines Produktes nicht dessen Sicherheit beweist und den Hersteller somit nicht von seinen moralischen oder rechtmäßigen Pflichten entbindet.

2.2 Arten der Zertifizierung

Zertifikate gibt es für unterschiedliche *Objekte*. Hier sollen die wichtigsten kurz vorgestellt werden. Die Zertifizierung von Systemen oder Produkten ist dabei die Wichtigste für diesen Artikel.

2.2.1 Organisationen oder Individuen

Das Ziel dieser Zertifizierung ist es, dass die beglaubigte *Organisation* ein bestimmtes Maß an Erfahrung aufweist, und die vereinbarten Standards erfüllt. Dabei ist es vergleichsweise leicht die Verfahren, die angewendet werden, auf Korrektheit zu überprüfen. Schwieriger wird es zu überprüfen, mit welcher Kompetenz diese durchgeführt werden.

Werden hingegen *Individuen* zertifiziert, wird diesen mittels des Zertifikates erlaubt, den gewählten Beruf in diesem Sektor auszuführen. Speziell im Fall von sicherheitskritischen Systemen ist ein hoher Grad an *Fachwissen*, *Erfahrung* und *Training* unter den Ingenieuren notwendig.
Als Beispiel sei hier die US Luftfahrtindustrie angeführt: Die FAA[7] autorisiert Angestellte der Herstellerfirmen, damit diese bestimmte Zertifizierungen innerhalb des Unternehmens durchführen. Bei *Boing* und *McDonnell Douglas* Projekten werden auf diese Art 90 - 95% aller Zertifizierungen durchgeführt. [2]

2.2.2 Werkzeuge oder Methoden

Die *Werkzeuge* und *Entwicklungsmethoden*, die in der Produktion von sicherheitskritischen Systemen eingesetzt werden, spielen eine wichtige Rolle, um dessen Leistung zu determinieren. Folglich setzen einige Standards Restriktionen, die die Nutzung von Werkzeugen und Methoden einschränken.
Besonders wichtig und weit verbreitet unter der Werkzeug-Zertifizierung ist die Zertifizierung von *Compilern*.

[7] Federal Aviation Authority (US Luftfahrtbehörde)

2.2.3 Systeme oder Produkte

Wie oben bereits erwähnt, kann die Zertifizierung von *Produkten* in bestimmten Bereichen der Industrie notwendig sein. Dies variiert allerdings stark von Anwendungsgebiet zu Anwendungsgebiet und von Land zu Land.
Als Beispiel sei hier die Zertifizierung von elektronischen Geräten in der Medizin angeführt. Diese ist in Großbritannien freiwillig, wohingegen sie in Deutschland und den USA verpflichtend ist.
Damit ein Produkt zertifiziert werden kann, muss es die Konformität mit dem zuvor festgelegten Standard beweisen. Die Zertifizierung von *Systemen* bezieht sich zumeist auf das komplette System und nicht auf einzelne Komponenten.

Die Zertifizierung von Software ist besonders schwierig, da hier ein zuverlässiges testen nur schwer möglich ist. Daher basiert die Zertifizierung sowohl auf dem *Entwicklungsprozess*, als auch auf der demonstrierten *Leistung* des Systems. Dazu müssen die angewendeten Methoden eingeschätzt werden, und eine Bewertung des Entwicklungsteams muss stattfinden. Unvermeidlich bilden also Aspekte der Qualitätssicherung einen grundlegenden Teil dieser Einschätzung.

2.3 Ziele der Zertifizierung

Nach diesem Überblick bleibt nun noch die Frage nach dem Zweck der Zertifizierung übrig. Als Ziele der Zertifizierung bei sicherheitskritischen Systemen sind die folgenden Punkte zu betrachten:

- Verbesserung der Sicherheit
- Steigerung des Bewusstseins über die Implikationen der Systemnutzung und Sicherheit
- Das Erzwingen von minimalen Standards in Design und Herstellung innerhalb der betroffenen Industrie
- Ermutigung des Aufbaus einer professionellen Verantwortlichkeits-Struktur

2.4 Safety Case

2.4.1 Einleitung

Der Safety Case ist ein *Dokument* oder eine Sammlung von Dokumenten, die *glaubhafte Argumente* präsentieren, dass das vorgeschlagene potentiell gefährliche System eine akzeptable Sicherheit aufweist.
Des Weiteren beschreibt der Safety Case alle *Risiken*, die bei der Nutzung des

Systems auftreten können, sowie die möglichen *Gefahren* beim Versagen des Systems. Weiterhin wird beschrieben, was getan werden kann, um die Wahrscheinlichkeit des Auftretens eines *Fehlers* zu minimieren. Der Safety Case dokumentiert die Sicherheit des Systems während seines gesamten *Lebenszyklus*.
Um den Safety Case auszuarbeiten, bedarf es in der Regel eines enormen Zeit- und Arbeitsaufwandes.

2.4.2 Zweck des Safety Case

Einer der wichtigsten Nutzen des Safety Case ist die Unterstützung der Zertifizierung, da der *Zertifizierer* nach Beweisen suchen wird, dass alle potentiellen Gefahren identifiziert und entsprechende Schritte veranlasst wurden, um diese zu behandeln. Genau diese Beweise liefert der Safety Case. Ebenso liefert er die benötigten Argumente, die aufzeigen, dass angemessene Entwicklungs- und Testmethoden eingesetzt wurden.
Allgemein hilft der Safety Case bei der Minimierung des *Projektrisikos*, denn potentielle Schwachstellen können so schon in einer frühen Projektphase erkannt werden.

2.4.3 Probleme bei der Entwicklung

Die meisten Safety Cases werden mit Hilfe von einfachen Werkzeugen entwickelt, verlassen sich aber auf *manuelle Prüfungen*, um ihre Konsistenz und Vollständigkeit zu versichern.
Das größte Problem bei der Entwicklung stellt die Tatsache dar, dass die Inhalte des Safety Case stets *multidisziplinär* sind. Das führt dazu, dass das Entwicklungsteam Experten aus verschiedenen Fachgebieten als Mitglieder haben sollte (vgl. Abbildung 2). Der Knackpunkt eines solchen Teams ist die Kommunikation zwischen den einzelnen Personen, denn diese benutzen zum Teil recht unterschiedliche Fachsprachen.

2.5 Der Prozess der System-Zertifizierung

Obwohl der *Zertifizierungsprozess* in der letzten Phase des Projektes durchgeführt wird, sollte er, genau wie die Verifikation, schon in einer *frühen Phase* mit in die Planungen einfließen.
Der Entwickler sollte schon frühzeitig Kontakt zum Zertifizierer aufnehmen, um Problemen aus dem Weg zu gehen, und das Verständnis zwischen den Parteien zu verbessern. Auf diese Weise kann der Zertifizierungsprozess erheblich erleichtert werden.

Während der Entwicklung muss der Entwickler den Zertifizierer mit Dokumenten

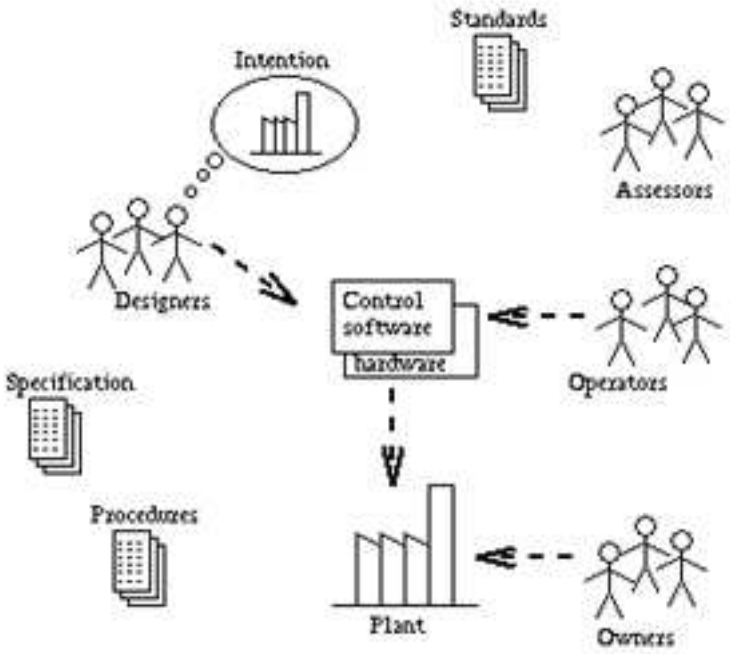


Abbildung 2: Expertengruppen bei der Entwicklung des Safety Case

versorgen, die darlegen, dass die Bedingungen des zuvor festgelegten *Zertifizierungsplans* eingehalten werden. Ebenso werden zu bestimmten Abschnitten des Projektes Daten zur Kontrolle vorgelegt. Daher kann man bei der Zertifizierung eher von einem *lang andauernden Prozess* sprechen, der parallel zur Entwicklung abläuft, als von einer einmaligen Phase am Ende des Projektes.

2.6 Zertifizierungsstandards

2.6.1 IEC 61508

Der erste Entwurf dieses Standards wurde unter der Bezeichnung IEC 1508 *Functional Safety: Safety-Related Systems* 1995 herausgegeben. Später wurde er in IEC 61508 umbenannt.
Der Standard beschäftigt sich primär mit sicherheitsrelevanten Systemen, die *elektrische*, *elektronische* oder *programmierbare elektronische Systeme* als Bestandteile enthalten. Außerdem gibt er generelle Anweisungen, die relevant für alle Formen von sicherheitskritischen Systemen sind, wie zum Beispiel die nötigen Elemente der Lebenszyklus-Phasen und der Dokumentation. Der Standard umfasst nicht nur die Kontrollelemente der Einrichtung, sondern auch alle externen risiko-mindernden Anlagen.
Er spezifiziert allerdings keine Individuen, die für die Umsetzung gewisser Auf-

gaben verantwortlich sind.
Ein Schlüssel-Element des Standards ist die Tatsache, dass es sich um einen *generischen Standard* handelt, der nicht auf einen bestimmten Industriebereich beschränkt ist.

2.6.2 DO-178B

Der DO-178B *Software Consideration in Airborne Systems and Equipment Certification* wird in den USA und Europa unter unterschiedlichen Bezeichnungen herausgegeben. So heißt er in den USA *RTCA*[8] *SC167/DO-178B* und in Europa *EUROCAE ED-12B*. Üblicherweise wird er jedoch schlicht mit DO-178B bezeichnet. Die Letzte Revision stammt aus dem Jahr 1992.
Obwohl es sich bei dem Standard um einen *industriespezifischen Standard* der zivilen *Luftfahrtindustrie* für die Entwicklung von Luftfahrt-Software handelt, wird DO-178B in einer Vielzahl von Branchen (z. B. Atomindustrie) als ein zu befolgendes Modell angesehen. Er beschäftigt sich, wie bereits erwähnt, ausschließlich mit Software. Zu seinen Kernbestandteilen gehören die *Software-Planung*, *-Entwicklung*, *-Verifizierung* sowie die *Software-Qualitätssicherung*.

Der Standard unterscheidet *fünf Zertifizierungsstufen* von Software, die sich nach den Folgeschäden eines Ausfalls bzw. einer Fehlfunktion richten:

- **Level A:** Katastrophale Folgen, z. B. der Absturz des Flugzeuges
- **Level B:** Ernsthafte Folgen, z. B. Tötung oder Verstümmelung einzelner Personen
- **Level C:** Erhebliche Folgen, z. B. Beschädigung von Eigentum
- **Level D:** Geringfügige Folgen, z. B. unnötiger Aufwand der Crew
- **Level E:** Keine Folgen

Als populäres Beispiel für den Einsatz von DO-178B in einer nicht Luftfahrt spezifischen Branche, sei hier das Betriebssystem *LynxOS* von LynuxWorks[9] angeführt. Dieses „Embedded-Operating System" ist ein gemäß DO-178B zertifiziertes, POSIX-kompatibels Echtzeit-Betriebssystem, das als Grundbaustein für ein später zertifizierbares Gesamtsystem dient.

[8] Radio Technical Commission for Aeronautics, http://www.rtca.org
[9] http://www.lynuxworks.com

2.6.3 Sonstige Standards

Neben diesen Standards existiert auch hier noch eine Reihe von anderen Standards im Zivil- und Militärsektor. Ein paar ausgewählte Standards aus diesen Sektoren sind die folgenden:

- Zivile Standards:
  - ESA PSS-05-0 *Software Engineering Standards* [11]
  - HSE Guidelines *Programmable Electronic Systems in Safety-Related Applications* [12]
  - IEC 880 *Software for Computers in the Safety Systems of Nuclear Power Stations* [13]
- Militär-Standards:
  - DoD Defence Standard 5200.28 *Trusted Computer Systems Evaluation Criteria* [14]

3 Schlussfolgerung

Abschließend bleibt anzumerken, dass *Qualitätsmanagement* und *Zertifizierung* eng miteinander verknüpft sind. Es ist unmöglich ein Zertifikat zu erhalten, ohne zuvor entsprechende Maßnahmen beim Qualitätsmanagement ergriffen zu haben. Beim Qualitätsmanagement und der Zertifizierung handelt es sich jeweils um *Prozesse*, die parallel zur *Entwicklung* und *Produktion* ablaufen.
Sicherheit befasst sich mit dem Ausschließen und Reduzieren des Risikos von Unfällen, und spielt daher auch ab einer sehr frühen Projekt-Phase eine wichtige Rolle. Der *Safety Case* ist einer der wichtigsten Bestandteile um die Sicherheit des Systems zu demonstrieren und zu dokumentieren.
Die Entwicklung von sicherheitskritischen Systemen sollte von gut organisierten, *multidisziplinären Teams* durchgeführt werden. Dabei sollten *Standards* eingehalten werden um am Ende ein möglicherweise benötigtes Zertifikat ohne großen Zusatzaufwand erwerben zu können.

Literatur

[1] *International Organization for Standardization (ISO)*, http://www.iso.ch

[2] Storey, Neil, *Safety critical computer systems*, Harlow, England 1996, Verlag: Prentice Hall

[3] Bennett, S., *Real-time Computer Control: An Introduction*, 1994, Verlag: Prentice Hall

[4] *Wikipedia - Six Sigma*, http://en.wikipedia.org/wiki/Six_Sigma, Stand: 30.03.2004

[5] *About ISO*, http://www.iso.ch/iso/en/prods-services/otherpubs/pdf/inbrief2002-en.pdf, Stand: 26.03.2004

[6] ANSI/IEEE Standard 730-1989, *Software Quality Assurance Plans*, New York 1989, Institute of Electrical and Electronic Engineers

[7] ASME Standard NQA 2a *Quality Assurance Requirements for Nuclear Facility Applications*, New York 1990, The American Society of Mechanical Engineers

[8] Department of Defence, Standard-2168 *Defense System Software Quality Program*, Washington DC 1988, Department of Defence

[9] Ministry of Defence, Defence Standard 00-16 *Guide to the Achievement of Quality in Software*, Glasgow 1984, Directorate of Standardization

[10] NATO, Quality Standard AQAP-150 *Requirements for Quality Management of Software Development*, NATO International Staff - Defence Support Division 1993

[11] ESA PSS-05-0 *Software Engineering Standards*, Paris 1991, European Space Agency

[12] HSE Guidelines *Programmable Electronic Systems in Safety-Related Applications, Vol. 1 & 2*, London 1987, Her Majesty's Stationary Office

[13] IEC International Standard 880 *Software for Computers in the Safety Systems of Nuclear Power Stations*, Genf 1986, International Electrontechnical Commission

[14] DoD Defence Standard 5200.28 *Trusted Computer Systems Evaluation Criteria*, Washington 1985, Department of Defence

[15] *Wikipedia*, http://en.wikipedia.org

[16] Giese, H., *Safety-Critical Computer Systems*, http://www.upb.de/cs/ag-schaefer/Lehre/Lehrveranstaltungen/Vorlesungen/SafetyCriticalComputerSystems/WS0203/index.html, Stand: 26.03.2004

[17] *Quality*, http://www.quality.de/

[18] Isaksen, Ulla; Bowen, Jonathan P.; Nissanke, Nimal *System and Software Safety in Critical Systems*, http://www.museophile.lsbu.ac.uk/pub/jpb/scs-survey-tr97.pdf, Stand: 26.03.2004

[19] *Validated Software*, http://www.validatedsoftware.com/what_is_DO178B.html, Stand: 26.03.2004

[20] *Functional safety and IEC 61508*, http://www.iee.org/oncomms/pn/functionalsafety/HLD.pdf, Stand: 26.03.2004

Risiko- und Gefahrenanalysen und deren Berücksichtigung beim Entwurf von Sicherheitskritischen Systeme

Seminararbeit
im Fach Informatik
im Rahmen des Seminars "**Sicherheitskritische Systeme**"
an der

Universität Siegen, Fachgruppe für Praktische Informatik

eingereicht bei
Dr. Jörg Niere

vorgelegt von
Kelen-Yo Rodrigue

Sommersemester 2004

Risiko- und Gefahrenanalysen und deren Berücksichtigung beim Entwurf von Sicherheitskritischen Systeme

1. Einleitung

Es ist heutzutage schon eine festgesetzte Tatsache bei der Entwicklung und der Produktion eines Systems eine bestimmte Methodik anzuwenden, zur Erleichterung und zum Überprüfen der Arbeitsschritte. Bei der Entwicklung sicherheitskritischer Systemen wird auch die ähnliche Arbeitsweise privilegiert. Sicherheitskritische Systeme sind Systeme bei denen ein Fehler dazu führen kann, dass das Leben einer Person gefährdet wird oder die Umwelt beschädigt wird. Es existieren viele gute bekannte Beispiele für sicherheitskritische Systeme in verschiedenen Anwendungsbereichen: medizinischen Geräten, Flugzeugsteuerungssystemen, Waffen- und Atomkraftwerksystemen. Manche modernen Informationssysteme sind allgemein gesehen sicherheitskritisch geworden, weil bei einem Fehler finanzielle Verluste und sogar lebensbedrohliche Zustände daraus resultieren können. Daher wird schnell ersichtlich warum die Entwicklung solcher Systemen nicht auf die leichte Schulter genommen werden kann. Daher muss die Analyse und der Design bei sicherheitskritischen Systemen sorgfältig durchgeführt werden, denn Hazards auch Gefahren genannt, können viele Menschen das Leben kosten, wenn es zum Unfall kommen würde. Die Risiko- und Gefahrenanalyse ist der Teil der Entwicklung wo das zu entworfenem System auf mögliche Gefahren und Risiko analysiert wird, das Ergebnis wird nach klassifiziert und unter Berücksichtigung bei dem Entwurf genommen, indem verschiedene Prinzipien angewendet werden, um das System auf Fehler zu reduzieren bzw. ganz auszuschließen, bis zufriedene Ergebnisse erreicht werden.
In dieser Seminararbeit wird außer der Definition von Begriffe ein konkretes Beispiel des Neigemoduls des Shuttles betrachtet. Das Beispiel stammt aus der Universität Paderborn wo an einem Projekt genannt SHUTTLE der Neuen Bahntechnik Paderborn gearbeitet wird.

2. Präsentation des Neigemoduls des Shuttles

Der Shuttle, der hier behandeln wird, gehört zu der Forschungsinitiative Neue Bahntechnik Paderborn. *Das Ziel ist die Kombination des herkömmlichen mechanischen Tragen und Führen auf dem bestehenden Schienennetz mit dem fortschrittlichen verschleißfreien Linearantrieb. Zusätzlich soll durch intelligente Fahrwerkstechnik ein höherer Fahrkomfort erzielt werden*[7]

Abbildung 1 Shuttle

2.1 Funktionsweise der Neigetechnik

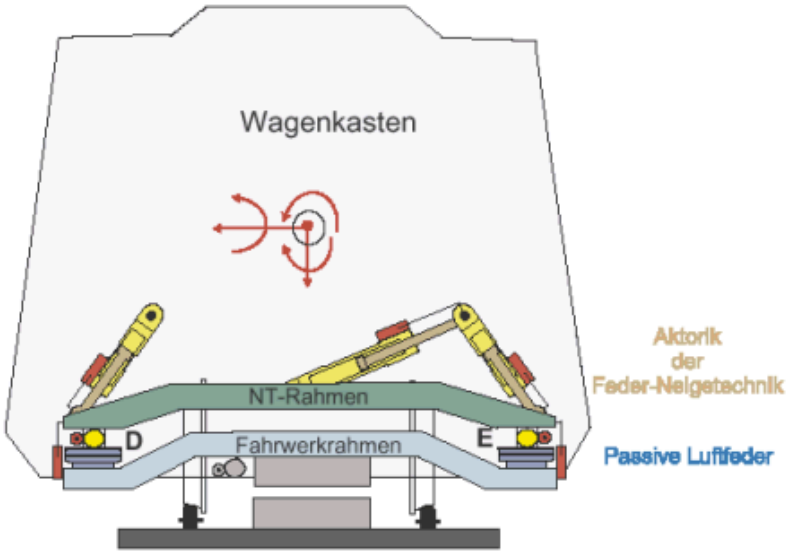


Abbildung 2 Wagenkasten

Das Feder- und Neigemodul ist eine Kombination einer niederfrequent abgestimmten passiven Aufhängung des Wagenkastens mittels einer Luftfeder (Sekundärfederung) mit einer aktiven Fußpunktverstellung, so dass der Wagenkasten aktiv gedämpft und geneigt werden kann. [7]

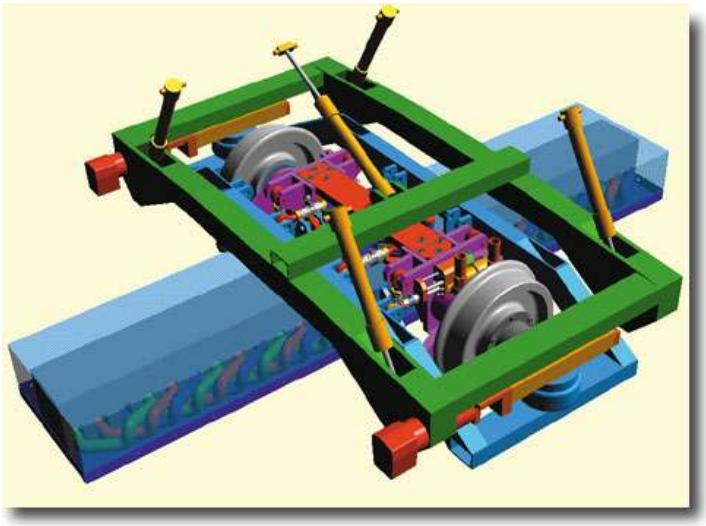


Abbildung 3 Neigetechnik

2.2 Problemstellung

Für die Regelung der Fußpunktverstellung, werden die benötigten Informationen von Sensoren zur Verfügung gestellt und mittels einer Mehrgrößenregelung verarbeitet. *Durch diese aktive Federungstechnik wird eine weitgehend dynamische Entkopplung des Wagenkastens, vor allem im oberen Frequenzbereich erreicht, d.h. die von den Unebenheiten in der Schiene (quer und vertikal) eingeleiteten Störungen werden so gut wie nicht mehr auf den Wagenkasten übertragen.* Es resultiert daraus, dass der Fahrkomfort in vertikaler und horizontaler Richtung bisher noch nicht erreicht wird.[7]

3. Systemanforderungen und Sicherheitsanforderungen

Bevor man zu Hazard- und Risikoanalyse geht, soll die Anforderungen an das System richtig

spezifiziert werden. Die Anforderungsspezifikation steht am Anfang jedes Entwicklungsprozesses und dient als Grundlage für die Realisierung eines technischen Systems, seine Bedeutung ist sehr groß. Bei sicherheitskritischen Systemen unterscheidet man zwei wichtige Arten von Anforderungen: Systemanforderungen und Sicherheitsanforderungen. Sie werden im Folgenden vorgestellt.

3.1 Systemanforderungen

Systemanforderungen sind die Anforderungen an das gesamte System, sie werden in funktionalen, nicht funktionalen und internen Anforderungen unterteilt.

1. Funktionale Systemanforderungen
    - Safety (Sicherheit)
    - Correctness (Korrektheit)
    - Trustability (Vertrauenswürdigkeit)
2. Nicht Funktionale Systemanforderungen
    - Reliability (Funktionsfähigkeit)
    - Availability (Verfügbarkeit)
    - Performance (Leistung)
    - Effizienz
    - Speicherplatz
3. Interne Systemanforderungen
    i. Entwicklung
        - Understandability (Verständlichkeit)
        - Testability (Testbarkeit)
        - Verifiability (Verifizierbarkeit)

    ii. Wartung
        - Recoverability (Wiederherstellbarkeit)
        - Maintainability (Wartbarkeit)
        - Extendability (Erweiterbarkeit)
        - Portierbarkeit

    iii. Wiederverwendbarkeit [4]

3.2 Sicherheitsanforderungen

Das System muss auch systemspezifische Sicherheitsanforderungen, die mit der Funktion und Eigenschaften des Systems verbunden sind, erfüllen. Um dies zu erreichen, wird eine Reihe von Aufgaben geleistet. Die Hauptstufen des Prozesses werden wie folgt gegliedert:

- Identifikation der mit dem System verbundenen Gefahren durch Gefahrenanalyse
- Klassifizierung der Gefahren nach ihrer „Schwere."
- Ermittlung von Methode um diese Gefahren zu mindern bzw. ganz auszuschließen.
- Zuordnung geeigneter Reliability und Availability Anforderungen zu diesen Methoden.
- Ermittlung eines geeigneten *Safety Integrity Level* und die dazu passende Methode auswählen.

Der Safety Integrity Level ist die Wahrscheinlichkeit, dass das sicherheitsrelevante System zufrieden stellend, die erforderlichen Sicherheitsfunktionen unter allen festgesetzten Bedingungen (Zuständen) innerhalb einer Zeitperiode durchgeführt.

4. Hazard-Analyse

Fast jeden Tag haben wir mit Maschinen zu tun, denen wir blind unser Leben anvertrauen. Es kann zu Hause, in der Straße oder am Arbeitsplatz sein. Wir stellen uns selten Fragen über ihre Sicherheit

und Zuverlässigkeit, weil es uns klar scheint, dass die Menschen, die diese Maschinen gebaut haben, an alles ausgedacht haben. Zum Beispiel fliegen wir in den Urlaub ohne uns viele Gedanke darüber zu machen, wie sicher das Flugzeug ist. In der Realität kann eine Fehlfunktion oder ein Bedienungsfehler, mit dem keiner gerechnet hat, zu tödlichen Folgen führen. Also bei der Entwicklung eines sicherheitsrelevanten Systems muss man für alles was geschehen kann, immer einer Ausweg bereithalten zum Verhindern, dass solche Fehler nicht eintreten.
Jede Kleinigkeit, die nicht vorher gesehen wurde, stellt eine latente Gefahr dar. Ein Prinzip bei den sicherheitsrelevanten Systemen ist:
"Ausfälle und Umstände, mit denen zu rechnen ist, dürfen nicht gefährlich auswirken" [1]
Das ist der Grund, warum es lebensnotwendig ist, so viele Gefahren wie möglich im Voraus zu identifizieren. An diesem Punkt spielt die Gefahrenanalyse eine entscheidende Rolle. Um das System auf potentielle Gefahren hin zu untersuchen, können viele Techniken benutzt werden, aber diese Techniken können nicht die menschliche Kreativität ersetzen.
Noch besonders wichtig ist zu wissen, dass die Analyse ein Prozess ist, der vor und während der Systementwicklung abläuft. *Jede Entwicklungsphase bringt neue Erkenntnisse und dadurch können neue Gefahren identifiziert werden.* Und es wird dabei ein Dokument mit dem Namen Gefahrenanalyse erzeugt, also mittels dieses Dokuments wird bei dem Entwurf des Systems versucht, alle Gefahren zu mindern oder zu eliminieren. Für ein klares Verständnis dieses Kapitels, werden einige Therme zuerst definiert. Es handelt sich um Gefahr und Fehler. Danach werden einige praktische analytische Techniken vorgestellt. Jeder Teil wird von einem Beispiel gefolgt.

4.1 Gefahr

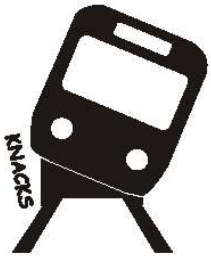
Hazard (Gefahr) ist eine Situation in welcher eine echte Bedrohung oder eine potentielle Bedrohung für Menschen und Umgebung besteht. Aus dieser Bedrohung kann ein Unfall entstehen. Es wird also eine Betrachtung der inneren und der äußeren Bedrohung des Systems durchgeführt. Die Untersuchung kann bei größeren Systemen sehr schwer sein, weil die Kombinationen von Ereignissen auch untersucht werden müssen. Es ist in der Wirklichkeit leider nicht immer möglich alles im Voraus zu sehen. Deswegen muss man bei jedem System mit einer verborgenen Rest-Gefahr rechnen.

4.2 Fehler

"Ein Fehler ist die Ursache für unerwünschte Resultate und kann bei den Menschen (Bediener) oder bei den Maschinen liegen. Aus der Umgebung können auch unerwartete Zustände kommen, die ebenso unerwünschte Resultate erzwingen." [1]

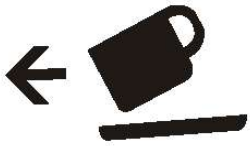
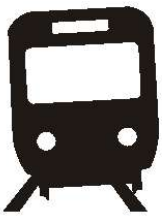
Klassifizierung der Fehler
Quelle: Entwicklungsfehler / Laufzeitfehler,
Art: Permanente / Sporadische / Bedingte Fehlertolerante,
Bereich: Wert / Zeit / unaufgeforderte Aktionen.

Beispiel: Hazard-Analyse des Neigemoduls des Shuttles
In dem folgenden Beispiel wird auf mögliche Hazards und ihre Folge gesucht.



1. Defekt des Chassis

- Aufhängung verbiegt sich
- Unvorhergesehene Kräfte auf Fahrzeugteile: Kette von Beschädigungen

2. Komfort Verlust 
    - Unwohlsein bei den Passagieren
    - Fliegende Objekte
    - Verletzungen
3. Entgleisung des Shuttles 
    - Sperrung und Beschädigung der Strecke
    - Personenschäden
4. Zusammenstoß durch zu starke Neigung mit Gegenständen
    - Personenschäden
    - Blockierung und Beschädigung der Strecke

4.3 Techniken der Analyse

Die Techniken der Hazard-Analyse existieren in verschiedenen Formen. Jede Technik bietet eine andere Einsicht in die Eigenschaften des Systems. Manche Methoden finden ihre Anwendung in bestimmter Situation und erreichen eine Grenze bei ihrer Nutzung in anderen Bereichen. Dabei kann man die Vielzahl dieser Technik in zwei Typen unterteilen: Vorwärts- und Rückwärts-Suche und Top-down und Bottom-up-Suche. Die Bezeichnungen der Typen, die in die nächsten Abschnitte tiefer betrachtet werden, sagen schon viel über deren Arbeitsstrategie, um die Gefahren zu analysieren.

4.3.1 Vorwärts- und Rückwärts-Suche

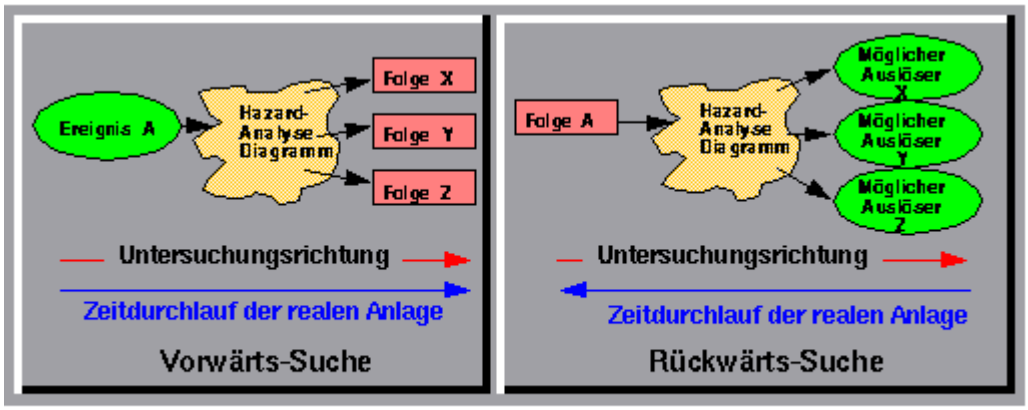


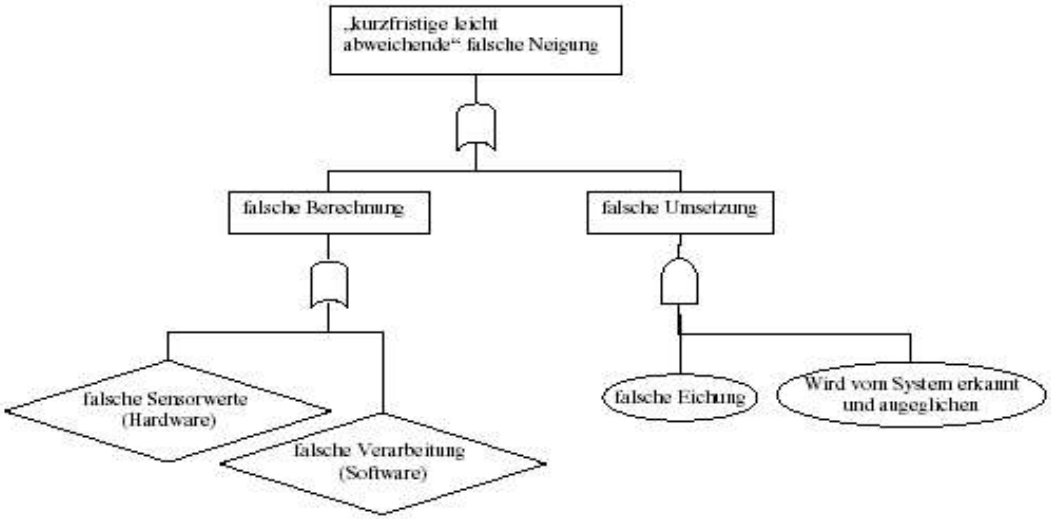
Abbildung 5 Vorwärts- und Rückwärts-Suche [1]

Bei der Vorwärts-Suche fängt man immer bei einem Ereignis an, und versucht die möglichen Folgen zu erwarten. Man nennt das auch induktive Suche, weil die Tätigkeit zeitlich nach vorn gerichtet wird.
Nun bei der Rückwärts-Suche geht man von einem gefährlichen Zustand aus und versucht herauszufinden, was die Ursache für diesen Zustand ist. Diese Methode wird als deduktive Suche bezeichnet. Zeitlich gesehen, im Gegensatz zur Vorwärts-Suche, wird nach hinten geschaut. [2]

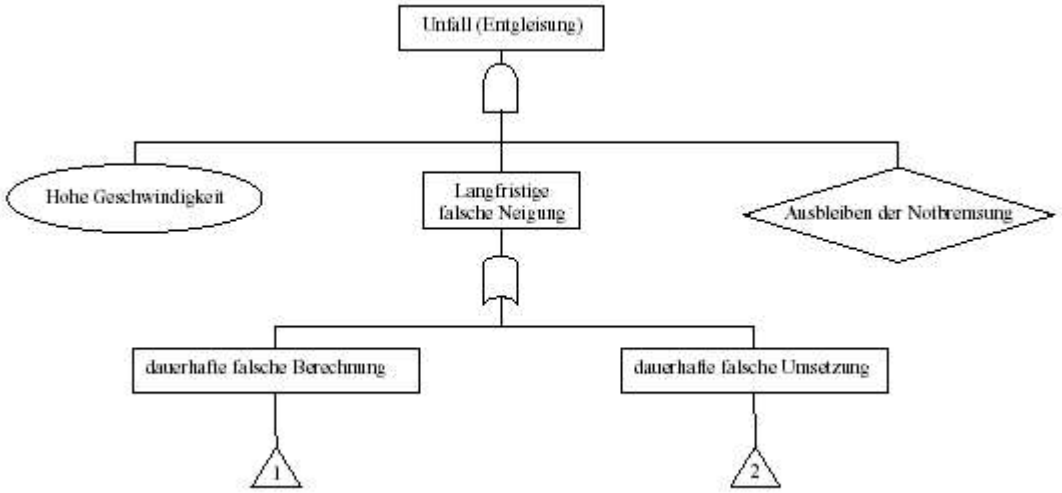
FTA (Fault tree analysis, auf Deutsch: Fehlerbaumanalyse)
Jetzt handelt es sich um ein Beispiel für die Rückwärts-Suche. Bei diesem Verfahren geht man von einer Gefahr aus und arbeitet Rückwärts, um die Ursache herauszufinden. Logische AND / OR Symbole charakterisieren die Beziehungen zwischen den Ereignissen. Es wird also beispielsweise durch ein AND-Symbol dargestellt, dass ein Ereignis dann eintritt, wenn zwei oder mehrere andere Ereignisse gleichzeitig stattfinden (Analog für OR). Zusätzlich wird es durch verschiedene Symbole zwischen elementaren Ereignissen (Kreis) und Fehlerereignissen unterscheidet, die durch andere Ereignisse hervorgerufen werden (Rechteck) sowie Ereignissen, deren Ursachen bislang noch ungeklärt sind (Raute). Der Prozess der Ursachenfindung und Dokumentation durch Boolescher Algebra wird solange fortgesetzt, bis die grundlegenden Ereignissen erreicht werden, die selbst keine Ursache haben bzw. deren Ursachen man nicht weiter betrachten möchte. [1]

<u>Beispiel aus dem Neigemodul des Shuttles</u>

a) Kurzfristige falsche Neigung



b) Langfristige falsche Neigung



4.3.2 Top-Down und Bottom-Up Suche

Bei der Bottom-Up Suche wird von dem Defekt einer kleinen Komponente ausgegangen und untersucht, welche übergeordneten Module dadurch getroffen werden. Es wird für jede Komponente im System diese Untersuchung durchgeführt, was einen sehr hohen Aufwand bedeuten kann. Wenn die Ausfälle kombiniert sind, kann das schlecht sein, weil diese Untersuchung keine Unterstützung für die Untersuchung von Kombinationen von Ausfällen bietet. [1]

FMEA (Failure modes and effects analysis)

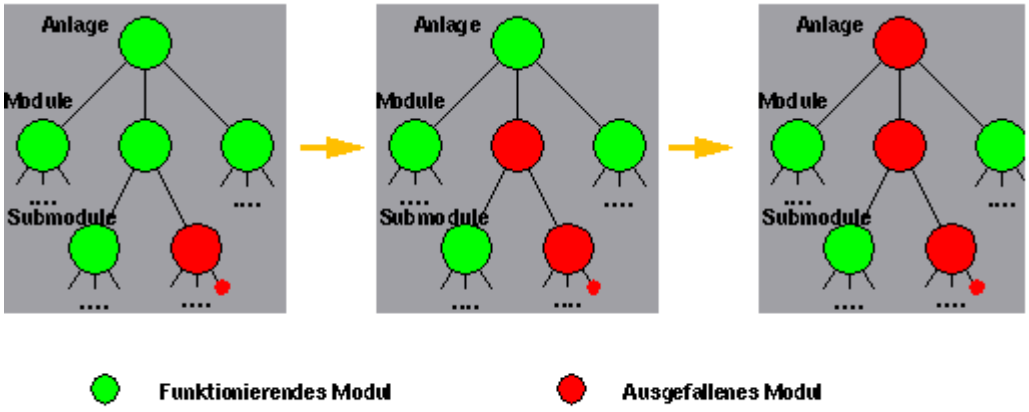


Abbildung 5 Fehlerfortpflanzung und Modulhierarchie [1]

Die FMEA bedeutet auf Deutsch "*Fehlermöglichkeiten und Einfluss-Analyse*". Sie wurde in den 60er Jahren bei der NASA und bei Appolo-Programm erstmals eingesetzt. Das Verfahren hat dank des Kraftwerksbaus einen breiten Eingang in die Automobilindustrie gefunden. Es handelt sich um eine präventive Qualitätssicherung. Dabei wird ein System detailliert auf möglichen Fehler und deren Ursachen und Wirkungen untersucht. Die Resultate werden aufgezeigt sowie Maßnahmen zur Fehlervermeidung und Fehlerentdeckung beschrieben. Da alle Komponentenfehler unter Berücksichtigt werden, eignet sich diese Annäherung besonders gut für Fälle, wo ein einziger Fehler zur gefährlichen Situation führen kann. Die Nachteile bei dieser Prozedur sind, dass keine Kombinationen von Ausfällen betrachtet werden können und die Analyse sich mit viele Ausfälle beschäftigen muss, die nicht zu Gefahren führen. Bei komplexen Systemen ist die Anwendung sehr teuer. Lösung ist FMEA, nur in einer späten Entwicklungsstufe anzuwenden, wo sie nur auf die

kritischen Bereiche angewandt wird. [1]

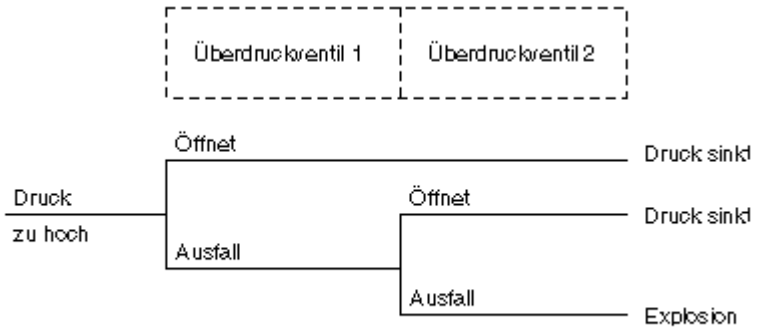


Abbildung 5 Beispiel eines ETA-Diagramms [1]

5. Risiko-Analyse

Nachdem im vorherigen Kapitel die Gefahrenanalyse vorgestellt wurde, wird in diesem Kapitel die Risikoanalyse vorgestellt. Das Risiko wird oft als die Kombination aus der Wahrscheinlichkeit des Eintreffens und den Folgen einer Gefahr verstehen. Das heißt:

Risiko = Wahrscheinlichkeit * Folgen

Das Risiko wird präsentiert als eine Quantifizierung der Gefahr. Auf den ersten Blick schein diese Definition handhabbar zu sein, aber wenn man tiefer analysiert, wird man schnell verstehen, dass Wahrscheinlichkeit und Folgen nur grobe und subjektive Abschätzungen sind. Alle Gefahren zu identifizieren, ist in der Realität eine schwere Sache und ebenso die Wahrscheinlichkeit zu bestimmen. "*Zweitens muss man eine Maßeinheit für die Schwere eines Unglücks finden und damit tatsächlich einen Preis für das Leben festlegen, was zumindest ethische Probleme mit sich bringt."* Daher ist es schwer die Frage nach einem akzeptablen Risiko mit Zahlen zu beantworten.

5.1 Nähere Betrachtung der Schwere der Folgen

Alle Industrien, die mit Sicherheit verwandt sind, klassifizieren Gefahren in Ausdrücke die ihre Schwere bezeichnen. Es ist offensichtlich, dass diese Klassifizierung von dem Industriezweig gemacht wird abhängt. Folgendes Beispiel aus militären Systemen im Großbritannien zeigt es uns besser:

Tabelle 2 Schwere der Unfallklasse für militäre Systeme

Klasse	**Definition**
Katastrophal	Mehrfache Tote.
Kritisch	Ein einzelner Tot, bzw. mehrfache schwere Verletzungen oder schwere berufliche Krankheiten.
Marginal	Eine einzelne schwere Verletzung oder berufliche Krankheiten, bzw. mehrfache untergeordnete(kleinere) Verletzungen oder untergeordnete(kleinere) berufliche Krankheiten.
Unwesentlich	Höchstens eine einzelne untergeordnete(kleinere) Verletzung oder untergeordnete(kleinere) berufliche Krankheiten.

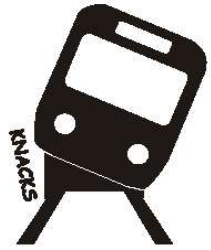
5.2 Nähere Betrachtung der Wahrscheinlichkeit

Die Wahrscheinlichkeit des Eintreffens eines gefährlichen Ereignisses kann in unterschiedlicher Art und Weise ausgedrückt werden, und wird qualitativ oder quantitativ gegeben. Manchmal wird die Rate als die Anzahl von Ereignissen pro Stunde oder pro Jahr (einer Operation) bezeichnet. Alternativ wird sie als die Anzahl der wahrscheinlichen eintreffenden Ereignisse während der Lebenszeit der Einheit verstanden.
Ein weiteres Beispiel aus dem Militär definiert sechs Klassen von Gefahrenwahrscheinlichkeit, die im folgende aufgezeigt werden.

Tabelle 2 Stufe der Unfallwahrscheinlichkeit für militäre Systeme

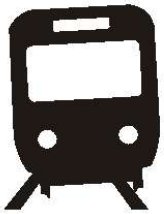
Unfallhäufigkeit	***Vorkommende Ereignisse während des Betriebs des Systems***
Häufig	Muss wahrscheinlich ständig experimentiert werden
Wahrscheinlich	Wahrscheinlich oft vorzukommen
Gelegentlich	Wahrscheinlich mehre Male vorzukommen
Entfernt	Wahrscheinlich ab und zu vorzukommen
Unwahrscheinlich	Unwahrscheinlich, aber wird außergewöhnlich vorkommen
Unglaublich	Extrem unwahrscheinlich, dass das Ereignis vorkommt

<u>Beispiel aus dem Neigemodul des Shuttles</u>
Risikoanalyse nach den Kriterien definiert in den zwei Tabelle:

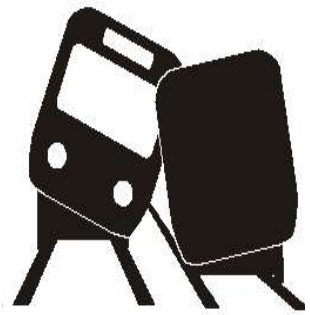


Zu 1. Marginal/Entfernt

Zu 2. Marginal/Wahrscheinlich



Zu 3. Critical/Entfernt



Zu 4. Katastrophal/Unwahrscheinlich

6. Entwurf

Es bestehen zwei grundlegende Verfahren des Entwurfs, die meistens nur in Kombination angewendet sollten. Das erste Verfahren ist die Benutzung von Entwurfmuster, das heißt die Benutzung von Standards und etablierte Code; das zweite ist die Benutzung von Informationen, die aus der Hazard Analyse gewonnen werden. In dieser Ausarbeitung wird der zweite Fall näher betrachtet und später für das Fallbeispiel des Neigemoduls des Shuttles eingesetzt wird.
Der Entwurf basierend aus der Hazard Analyse legt vier Designprinzipien fest. Diese Designprinzipien sind in ihrer Reihenfolge einzuhalten, die Anwendung einzelner ist unvollständig, um alle Gefahren von vornherein zu beseitigen. Diese Designprinzipien, die den nächsten Unterkapiteln erklärt werden, sind: „Hazard elimination“, „Hazard reduction“, „Hazard control“ und „Damage minimization“. [5]

6.1 Hazard elimination

Ziel hier ist das System so weit wie möglich sicherer zu gestalten, durch das Anwenden von mehreren Techniken. Der Ausgangspunkt ist die Komplexität des Systems zu vereinfachen, was die Überschaubarkeit und die Verifizierbarkeit erhöht, es sollten auch statt monolithische Sprache objektorientierte Programmiersprache ausgewählt. Der weitere Punkt ist die Entkopplung von Systemteile, das reduziert ungewollte Interaktionen und ermöglicht eine leichte Gruppierung und Eingrenzen der systemkritischen Funktionen. Schließlich kommt die Eliminierung von menschlichen Fehlern und die Reduzierung gefährlicher Bedingungen. [5]

Beispiel aus dem Neigemodul des Shuttles

a) Geschwindigkeitsreduzierung
   Löst z.B. den Hazard „Komfortverlust“
b) Ausreichender Abstand zu „Hindernisse“ neben der Bahnstrecke (Tunnel, Brückenpfeiler, Parallelgleis ...)
c) Absicherung der Gepäckstücke im Inneren.
d) Ausschalten der „Neigetechnik“
   Shuttle kann sich nun nicht mehr in Gegenspur neigen
   Hazard des „ Komfortverlustes“ bleibt.

6.2 Hazard reduction

Unfälle sind unwahrscheinlich, je unwahrscheinlicher der dafür verantwortliche Hazard ist, die Wahrscheinlichkeit für einen Hazard kann man senken, indem die Bedingungen, die zu einem Hazard führen, reduziert werden. [6]
Dieser Teil hat viel mit der Software zu tun, also sie wird so entwickelt, dass die Wahrscheinlichkeit des Eintreten eines Hazard auf mehrere Weisen reduziert wird. Es werden passive und aktive *Safeguards* eingesetzt; *Monitoring* für kritische Funktionen mit Hilfe von Entwurfmuster; *Lockouts,* um den Zugang zu kritischen Prozessen oder Zuständen zu erschweren;

Lockins, um das Verlassen eines sicheren Zustandes zu erschweren; und Interlocks, die wie Semaphore funktionieren. Weitere Verfahren sind die Benutzung von *Redundanzen*, um zum Beispiel das System von willkürlichen Hardwarefehlern zu schützen, und die *Wiederherstellung* der älteren Daten im Falle eines Fehlers. [6]

Beispiel aus dem Neigemodul des Shuttles

a) Model-Checking der Software
b) Sensoren an Fahrzeugende und -spitze
   Anbringen von Redundanten Sensoren
c) Möglichst lokale Überprüfung von Aktoren und Sensoren
   Kontrollsoftware direkt an Sensoren und Aktoren, die Fehlermeldung an Zentralsoftware senden kann

6.3 Hazard control

Es gibt Hazards, die „nötig" sind, weil sie nur schwer auszuweichen sind. Wenn das der Fall ist, muss ein solcher Zustand nur so kurz wie möglich besucht werden, das heißt die Aufenthaltszeit in einem unsicheren Zustand muss möglichst reduziert werden. Um das zu erreichen, gibt es der *Fail-Safe-Design.* Dies ist Charakterisiert durch so genannte *Fallback*-Zustände, in denen das System zurückkehren kann sobald ein riskanter Zustand entdeckt wird. Die bekannteste *Fallback*-Zustände sind das partielle Shutdown, mit minimaler Funktionalität und das Not-Shutdown, das keine Funktionalität gewährleistet, da das System einfach heruntergefahren wird. Die Alternative zu *Fallback*-Zustände ist das Ausstatten des System mit extrem sicheren Hazard-Detektoren. [6]

Beispiel aus dem Neigemodul des Shuttles

Fallback:
   Einrichtung von geeigneten Fallback-Ebenen
   Zentralsoftware hat kurze „Modulwege" zu verschiedenen Fallback-Ebenen

6.4 Damage minimization

Ziel dieses Abschnitts ist das Reduzieren des Schaden nach einem Unfall, wenn das passieren sollte, denn das System an sich und somit auch die Software nichts mehr machen kann. Es muss also ein „point of no return"definiert werden. Dieser bestimmt ab wann ein Hazard unausweichlich ist. [6]

Beispiel aus dem Neigemodul des Shuttles

a) Extra Verstärkung des Chassis
b) Nicht bzw. schwer entflammbare Materialien
c) Schnelle Vororthilfe

7. Zusammenfassung

Bei sicherheitskritischen Systemen spielt die Hazard- und Risikoanalyse eine schwerwiegende Rolle, sie ist unumgänglich für die Entwicklung des Systems. Diese Analyse muss so gut wie möglich gemacht werden, weil das Ergebnis unter Berücksichtigung für den Entwurf genommen wird. Gemäß der Erfahrung hat sich gezeigt, dass Fehler bei der Hazard-Analyse und bei dem Design, die nur nach der Entwicklung des Systems, entdeckt werden, sehr großer Aufwand verursachen können. Das ist der Grund warum alles sehr genommen werden muss.
Aus dem Beispiel wird erkannt, dass geeignete Maßnahmen im Design die Wahrscheinlichkeit des Vorkommens eines Hazards vermindern oder ausschließen können, aber alle Gefahren lassen sich nicht ausschließen.
Dieser Bereich der Informatik ist einen Zukunftweisenden Bereich, auf Grund der immer steigenden

Anzahl von eingebetteten Softwaresysteme in der Industrie.

8. Literatur

[1] www.first.fhg.de/~sergio/public/hazard.html
[2] www.theoinf.tu-ilmenau.de/ISCRA/hazard.html
[3] Safety-Critical Computer Systems, Neil Storey, Addison Wesley Longman 1996
[4] http://wwwcs.upb.de/cs/ag-schaefer/Lehre/Lehrveranstaltungen/Seminare/AEIzS/y
[5] http://wwwcs.upb.de/cs/ag-schaefer/Lehre/Lehrveranstaltungen/Seminare/AEIzS/f
[6] http://wwwcs.upb.de/cs/ag-schaefer/Lehre/Lehrveranstaltungen/Seminare/AEIzS/Abgaben/Folien/8_Entwurfsmuster_ASeibel.pdf#
[7] http://www.railcab.de/

Untersuchung der Programmiersprache Ada zur Implementierung sicherheitskritischer Systeme

Seminararbeit
im Fach Informatik
im Rahmen des Seminars "**Sicherheitskritische Systeme**"
an der

Universität Siegen, Fachgruppe für Praktische Informatik

eingereicht bei
Dr. Jörg Niere

vorgelegt von
Peter Sakwerda

Sommersemester 2004

Inhalt

1. Die Entstehung der Sprache....3
1.1. Motivation....3
1.2. Die Umsetzung....3
1.3. Ada s Anwendungsbereiche....4
2. Anforderungsspezifikationen an eine Programmiersprache zur Implementierung von sicherheitskritischer Systeme....4
2.1. Allgemeine Anforderungen an SkS....4
2.2. Allgemeine Anforderungen an eine Programmiersprache (nach *Carré*)....5
2.3. Probleme gewöhnlicher Programmiersprachen (nach *Clutterbuck*)....5
2.4. Charakteristische Merkmale einer Prog.sprache (nach Cullyer)....5
2.5. Anforderungskatalog an eine Programmiersprache für SkS....6
3. Umsetzung der SkS Merkmale in Ada ....
3.1. Allgemeine Merkmale von Ada ....
3.2. Reliability (Funktionsfähigkeit) ....
3.2.1. sichere Datentypen....
3.2.2. willkürliche Sprünge (wild Jumps)....
3.2.3. Überschreibung (overwrites) ....
3.3. Availability (Verfügbarkeit)....
3.3.1. logische Gültigkeit(logical soundness) ....
3.3.2. Initialisierungsfehler(initialilising failure) ....
3.3.3. Ausdrucksauswertungsfehler (expression evaluation error) ....
3.4. Performance (Leistung) ....
3.5. Zuverlässigkeit (Dependability) ....
3.5.1. Mehrfache Referenzen (aliasing)....
3.5.2. Seiteneffekte durch Unterprogramme (subprogram side-effects) ....
3.5.3. Speicherüberwachung (exhaustion of memory) ....
3.6. Understandability (Verständlichkeit)....
3. . Testability (Testbarkeit), Verifiability (Verifizierbarkeit)....
3. . Recoverability (Wiederherstellbarkeit)....
3. .1. Arithmetische Laufzeitüberwachung (operational arithmetic)....
3. . Maintainability (Wartbarkeit), Extendability (Erweiterbarkeit), und Wiederverwendbarkeit....10
3.10. sicheres Subset....10
4. Zusammenfassung....11
4.1. Vorwort....11
4.2. Beurteilung von Ada....11
5. Anhang....12
5.1. Literaturverzeichnis 12

1. Die Entstehung der Sprache

1.1. Motivation

Die Geschichte von Ada begann 1 4. Das amerikanische Verteidigungsministerium (U.S. Department of Defense, DoD) lies den Aufwand für die zukünftige Erstellung und Pflege von Programmsystemen im Bereich des Verteidigungswesen abschätzen.
Die Kosten sind auf über 3 Milliarden US-Dollar jährlich taxiert worden, eine Summe die noch für heutige Zeiten, dreißig Jahre später, enorm ist.
Hauptursachen für die immensen Kosten waren:

- Im Bereich des Verteidigungsministeriums waren, aufgrund vieler unterschiedlicher Hardware Plattformen, Hunderte verschiedene Programmiersprachen im Einsatz. Mehr als die hälfte der betreuten Software war für eingebettete Systeme bestimmt, die in speziellen Umgebungen zur Einsatz kommen wie z.B.: Sensor und Waffentechnik.

- Die entstandene Spezialisierung benachteiligte das DoD bei der Auftragsvergabe, da im Falle einer eingeführten Systemumgebung die Auswahl der möglichen Lieferanten klein, meistens auf einen Anbieter begrenzt, war. So konnte oft der Anbieter den Preis bestimmen, anderseits konnten die Entwicklungskosten für eine Spezialumgebung nur auf ein Projekt umgelegt werden und nicht auf mehrere verschiedene Anwendungen (Projekte).

- Der größere Teil der verwendeten Programmiersprachen ist einfach veraltet gewesen, die Neuerstellung von Software war teilweise teuerer als die Wartung.

1 5 wurde innerhalb der DoD eine Kommission (Higher Order Language Working Group kurz HOLWG) gegründet, deren Aufgabe war die Möglichkeit der Vereinheitlichung der Programmierwerkzeuge zu untersuchen.
Die Kommission stellte in mehreren Schritten Anforderungskataloge auf, untersuchte bereits vorhandenen und eingeführte Sprachen am Maßstab der Anforderungen, und kam zu dem Ergebnis das keine der untersuchten Programmiersprechen über alle erforderlichen Merkmale verfügt. Es hat sich dabei aber rausgestellt das es möglich sei eine neue Sprache zu Entwickeln die Eine volle Realisierung der Anforderungen darstelle, so wurde der Grundstein für die Entstehung von ADA gelegt.

1.2. Die Umsetzung

Als man 1 6/ aus den gestellten Anforderungen an die neue Programmiersprache eine formale Sprachspezifikation aufgestellte, hat das DoD die Entwicklung einer, dieser Spezifikation entsprechenden Sprache international ausgeschrieben. Insgesamt wurden 1 Sprachvorschläge eingereicht, von denen wurden erst mal 4 weiterentwickelt.
Nach nochmaliger Beurteilung im März 1 kamen zwei Entwicklerteams weiter, die abschließende Entscheidung fand März/April 1 statt. Die französische Arbeitsgruppe unter Leitung von Jean Ichbiah von CII-Honeywell wurde zum Sieger erkoren, die Sprache erhielt den Namen Ada benannt nach Ada Augusta, Countess of Lovelace, die im 1 Jahrhundert die ersten theoretisch lauffähigen Programme für die von Charles Babbage konstruierte Analytische Maschine geschrieben hatte, und damit die erste Programmiererin war.
Die endgültige Sprachdefinition legte das US-Verteidigungsministerium 1 0 fest.
1 3 wurde der ANSI-Standard (Ada 3), 1 4 der ISO-Standard vergeben [Kra 6].

1.3. Ada s An endungsbereiche

Ada wird hauptsächlich im militärischen Bereich und in der Raum- und Luftfahrt eingesetzt.
1 3 wurde *Ada* als einzige Programmiersprache für defense mission-critical applications [2] definiert.
1 6 wurde *Ada* zur einzigen Programmiersprache für alle C I-Systeme der NATO.
Um ein Beispiel aus dem zivilen Bereich zu nennen ist die Software der Boeing zu , in ADA geschrieben.

Grundsätzlich ist Ada besonders zu empfehlen bei der Erstellung von großen, langlebigen Softwareprojekten.

2. Anforderungsspezifi ationen an eine Programmiersprache zur Implementierung von sicherheits ritischer S steme

2.1. Allgemeine Anforderungen an S S

Die allgemeinen Anforderungen an sicherheitskritische Systeme sind laut [Loe04] wie folgt definiert:

- **Fun tionale S stemanforderungen**
  - Safety (Sicherheit)
  - Correctness (Korrektheit)
  - Trustability (Vertrauenswürdigkeit)

- **Nicht Fun tionale S stemanforderungen**
  - Reliability (Funktionsfähigkeit)
  - Availability (Verfügbarkeit)
  - Performance (Leistung)
  - Zuverlässigkeit (Dependability)

- **Interne S stemanforderungen**
  - Entwicklung
    - Understandability (Verständlichkeit)
    - Testability (Testbarkeit)
    - Verifiability (Verifizierbarkeit)
  - Wartung
    - Recoverability (Wiederherstellbarkeit)
    - Maintainability (Wartbarkeit)
    - Extendability (Erweiterbarkeit)
  - Wiederverwendbarkeit

Die funktionalen Systemanforderungen sind erst mal für die Untersuchung von Ada nicht von primärer Bedeutung, da sie das Gesamtsystem beschreiben. Bedeutungsvoll sind die nichtfunktionalen und internen Systemanforderungen. Sie beschreiben einzelne Anforderungen die alle Systemkomponenten im einem sicherheitskritischen System erfüllen sollten sei es Soft- oder Hardware. Für die Untersuchung einer Programmiersprache sind die Anforderungen etwas zu grob klassifiziert, sie müssen etwas feiner, unter Berücksichtigung Softwaretechnischer Gesichtpunkte, graduiert werden.

[2] Militärischen Angaben laut der Webseite: http://page.mi.fu-berlin.de/ vratisla/Militinf/militinf.htm

2.2. Allgemeine Anforderungen an eine Programmiersprache (nach *arr*)

Carr nannte 1 0[Carr 0] sechs Kriterien, die für die Eignung bzw. den Einsatz von Programmiersprachen in hoch integrierten Systemen besonders wichtig sind. Diese Kriterien sind 1 6 durch [Sto 6] wie folgt zusammen gefasst worden:

- logische Gültigkeit (*logical soundness*)
- einfache Definitionen (*simple Definitions*)
- Ausdruckstarker Code (*expressive power*)
- Sicherheit (*sequrity*)
- Verifizierbarkeit (*verifiability*)
- Zeit /Speicher Bedarf (*bounded space and time requirements*)

Keine der üblichen, im großen still eingesetzten, Sprachen erfüllt all diese Kriterien. Der Grund ist relativ einfach: die sicherheitskritische Software repräsentiert nur einen Bruchteil der weltweit vertretenen Software. Weiterhin wiedersprechen sich einige Kriterien, wie z.B. wenn eine Sprache einen sehr ausdrucksstarken Code hat tendiert sie sehr Komplex zu werden, das erschwert die Verifizierung erheblich. Die Entwickler der Sprache sind also gezwungen Kompromisse zu finden damit möglichst alle Punkte optimal berücksichtigt werden.

2.3. Probleme ge hnlicher Programmiersprachen (nach *lutterbuck*)

Über die Jahre konnte die Fachwelt viele Erfahrungen über die Zuverlässigkeit der Systeme und über die Art und vor allem die Ursachen von Fehlern sammeln die bei den meisten Programmiersprachen auftreten.
1 2 listete Clutterbuck [Sto 6], die vier meist vorkommenden Fehler in Programmen, die mit unterschiedlichen Sprachen geschrieben worden:

- Seiteneffekte durch Unterprogramme (*subprrogram side-effects*)
- Mehrfache Referenzen (*aliasing*)
- Initialisierungsfehler (*initialising failures*)Auswertungsfehler (expression evaluation errors)

2.4. Chara teristische Mer male einer Prog.sprache (nach Cull er)

Cullyer untersuchte 1 0 mehrere Programmiersprachen[3] mit dem Ziel, rauszufinden welche der gegebenen Sprachen sich besonders für den Einsatz in hoch-integrierten Systemen eignet.
Er differenzierte einige ausgewählte Programmiersprachen nach folgenden Faktoren:

- willkürliche Sprünge *(wild Jumps)*
- Überschreibung (overwrites)
- eindeutige Semantik (semantics)
- strenges mathematisches Modell (model of maths)
- arithmetische Laufzeitüberwachung (operational arithmetic)
- sichere Datentypen (data typing)
- Ausnahmenbehandlung (exception handling)
- sicheres subset (safe subset)
- Speicherüberwachung (exhaustion of memory)
- separate Paketübersetzung (separate compilation)
- gut verständlich (well understood)

[3] es war mir leider nicht möglich, sein Artikel The Choice of Computer Languages for use in Safety-Critical Systems zu beschaffen

2.5. Anforderungs atalog an eine Programmiersprache f r S S

Diese Merkmale zusammengefasst erstellen ein Anforderungskatalog, mit welchem die Programmiersprache Ada bezüglich der Eignung für den Einsatz in sicherheitskritischen Systemen untersuchen werden kann.

1. **Reliabilit (Fun tionsfähig eit)**
   - sichere Datentypen (data typing)
   - willkürliche Sprünge (wild Jumps)
   - Überschreibung (overwrites)

2. **Availabilit (Verf gbar eit)**
   - logische Gültigkeit (logical soundness)
   - Initialisierungsfehler (initialising failures)
   - Auswertungsfehler (expression evaluation errors)

3. **Performance (Leistung)**
   - Zeit /Speicher Bedarf (bounded space and time requirements)

4. **uverlässig eit (Dependabilit)**
   - Seiteneffekte durch Unterprogramme (subprrogram side-effects)
   - Speicherüberwachung (exhaustion of memory)
   - Mehrfache Referenzen (aliasing)

5. **Understandabilit (Verständlich eit)**
   - einfache Definitionen (simple Definitions)
   - Ausdruckstarker Code (expressive power)

6. **Testabilit (Testbar eit) Verifiabilit (Verifizierbar eit)**
   - eindeutige Semantik (semantics)
   - strenges mathematisches Modell (model of maths)

. **Recoverabilit (iederherstellbar eit)**
   - arithmetische Laufzeitüberwachung (operational arithmetic)
   - Ausnahmenbehandlung (exception handling)

. **Maintainabilit (artbar eit) Extendabilit (Er eiterbar eit) iederver endbar eit**
   - separate Paketübersetzung (separate compilation)

. **sicheres subset (safe subset)**

Zusätzlich muss eine Programmiersprache Werkzeuge zur Verfügung stellen die den Entwicklern z.B. Verwaltungsarbeit abnehmen, und ihn sonst mit hilfreichen features zur Seite stehen. Eine große Nutzergemeinschaft ist ebenfalls von essentieller Bedeutung, zur einen werden Fehler in der Entwicklungsumgebung, hauptsächlich in den Compilern eher entdeckt und beseitigt. Zum anderen wird durch Erfahrungsaustausch ein Expertenwissen aufgebaut der gute Code liefert.

3. Umsetzung der S S Mer male in Ada

Im folgenden werden die einzelnen Punkte aus dem Anforderungskatalog, untersucht und deren Umsetzung in Ada erläutert.

3.1. Allgemeine Mer male von Ada

Die Programmiersprache Ada wird durch die im Kapitel 1.2 bereits erwähnten Standards definiert. Eine **Validierung** eines Ada-Compilers gegen den Ada Sprachstandard ist notwendig, sonst darf man den Namen Ada nicht verwenden, weiterhin sichert die Validierung die **Portierbar eit** zwischen verschiedenen Compilerherstellern und Hardwareplattformen.
Ada gehört zu den Programmiersprachen über die Objektorientierung und andere moderne Software Engineering Methoden wie Datenkapselung und Information Hiding verbreitet wurden.
Zudem beherrscht Ada **Multitas ing**, innerhalb eines Programms können mehrere Tasks definiert werden, die als selbständige Threads laufen. Dadurch kann man elegant die Behandlung von asynchronen Ereignissen und Datenströmen programmieren.
Ada hat definierte Schnittstellen zu anderen Programmiersprachen wie z.B.: C, FORTRAN und COBOL. Dadurch lassen sich vorhandene Komponenten elegant in ein Programm einbinden.

3.2. Reliabilit (Fun tionsfähig eit)

3.2.1. sichere Datent pen

Ada ist eine streng typisierte Sprache, implizierte Typkonvertierungen finden nicht statt. Datenobjekte werden durch ein Typsystem kategorisiert, für jede Operation ist genau festgelegt, welchen Typs die Operanden sein dürfen und welchen Typs das Ergebnis sein wird. Das vorgegebene Typsystem kann durch den Programmier nach Bedarf erweitert werden, solange die gültigen Konventionen eingehalten werden.

3.2.2. ill rliche Spr nge (ild umps)

Ada kennt einen GOTO Befehl und lässt willkürliche, unkontrollierte Sprünge (wild Jumps) zu. Mit einem Sprung kann eine Anweisungsfolge verlassen oder die Kontrolle innerhalb einer Anweisungsfolge weitergegeben werden. Externe Sprünge in eine Anweisungsfolge, wie z.B. in den Rumpf einer Schleife oder in einen der Zweige einer IF-Anweisung, sind nicht möglich.
Vom Einsatz von Sprungbefehlen in sicherheitsrelevanten Softwaresystemen ist dringen abzuraten, sie sind für den Leser eines Programms kaum nachvollziehbar.
Anmerkung des Autors:
In der Literatur befinden sich widersprüchliche Angaben, laut [Sto 6] sind unkontrollierte Sprünge nicht möglich, in der Ada 5 Reference Manual von Gnat wird aber die *GOTO* – ANWEISUNG ausführlich erläutert.

3.2.3. berschreibung (over rites)

Das Speichermanagement von Ada, bietet einige Funktionen, um die Datenkonsistenz im Speicher zu gewährleisten. Laut [Sto 6] lassen sich diese jedoch umgehen, was wiederum zur Fehlfunktionen führen könnte.

3.3. Availabilit (Verf gbar eit)

3.3.1. logische G ltig eit(logical soundness)

In ihren Grundzügen ist Ada sehr klar und logisch strukturiert, was für den Einsatz in SkS von essentieller Bedeutung ist. Ada beherrst jedoch auch Konzepte der modernen Softwaretechnik wie Überladung, Vererbung und Polymorphismus. Diese komfortablen Sprachkonstrukte schaffen ein hohes Maß an Flexibilität und Abstraktion, steigert aber auch enorm die Komplexität der Programme, was wiederum zu Folge hat, dass die logische Gültigkeit der Anweisungen für den Entwickler nicht immer erkennbar ist.

Grundsätzlich ist es empfehlenswert diese Konstrukte in SkS nicht einzusetzen.

3.3.2. Initialisierungsfehler(initialilising failure)

Grundsätzlich ist ein Initialisierung der Datenobjekte optional. Deshalb ist es im herkömmlichen Ada nicht erkennbar, ob ein skalarer Wert korrekt vorbelegt wurde. Die Verantwortung für die Initialisierung wird aber nicht komplett den Programmier überlassen, auf Wunsch wird dies durch ein entsprechendes Pragma[4] vom Laufzeitsystem übernommen. Das Pragma *Normalize_Scalare* belegt alle Skalarwerte mit einer Vorbelegung, die nach Möglichkeit **nicht** dem legalen Wertebereich des Skalartyps entnommen ist. Auf diese Weise kann später sicher, und nicht aufgrund eines Zufalls, erkannt werden, ob eine Vorbelegung überhaupt erfolgt ist. Zu diesem Zweck steht ein Attribut *Valid* zur Verfügung, dass sie legale Vorbelegung der Skalare verifiziert.

3.3.3. Ausdruc saus ertungsfehler (expression evaluation error)

Die Laufzeitüberwachung (Ausnahmenmechanismus) überwacht die korrekte Ausführung der Ausdrücke. Bei unerwarteten Ereignissen, wird eine Ausnahme ausgelöst, die durch eine Behandlungsroutine abgefangen wird.

3.4. Performance (Leistung)

Performance einer Programmiersprache lässt kann eigentlich nur im Vergleich zu einer anderen Programmiersprache ausdrücken. Ada ist im Vergleicht z.B. zum C-Code eher langsam, die Ursache hierfür ist, dass Ada neben der eigentlich zu realisierenden Funktion auch permanente und umfassende Laufzeitüberprüfungen bereit stellt. Der Vollständigkeit halber sei erwähnt, dass die Ada-Compiler fast immer hochoptimierend sind und sehr effizienten Maschinencode liefern [Kra 6].

3.5. uverlässig eit (Dependabilit)

3.5.1. Mehrfache Referenzen (aliasing)

In Ada können dynamische Datenstrukturen (Objekte) erzeugt werden. Auf diese Strukturen kann durch sogenannte Zugriffsobjekte bzw. Zeigerobjekte zugegriffen werden. Mehrere Zugriffsobjekte können auf die gleiche dynamische Struktur verweisen, im Klartext: in Ada ist aliasing möglich.

3.5.2. Seiteneffe te durch Unterprogramme (subprogram side-effects)

Unterprogramme werden in Ada entweder durch Prozeduren oder durch Funktionen realisiert. Bei der Parameterübergabe der Prozeduren unterscheidet man drei Arten der Übergabe: *IN, OUT* und *IN OUT.* Die reine Wertvorbelegung (Übergabeart *IN*) stellt kein Problem dar, hier wird lediglich eine Kopie des Wertes der Prozedur übergeben. Bei den Übergabearten *OUT* und *IN OUT*, ist ein Schreibzugriff auf den Übergabeparameter möglich. Der geänderte Parameter wird nach Beendigung der Prozedur an den Aktualparameter aus dem Aufruf rückübermittelt, es können also Seiteneffekte auftreten. Der Gebrauch von Funktionen ist dagegen unkritisch weil bei den Funktionen nur die Übergabeart *IN* für die Parameterübergabe zugelassen ist, und die kann, wie schon bereits beschrieben, keine Seiteneffekte auslösen.

[4] Bestandteil von Annex H: Safety and Security

3.5.3. Speicher ber achung (exhaustion of memor)

Die Speicherüberwachung in Ada übernimmt der Ausnahmenmechanismus. Alle Probleme mit der Speicherzuweisung werden der Ausnahme *Storage_Error* zugeordnet. Die Ausnahme kann vornehmlich bei Speicherallokierung durch den *NEW*-Operator auftreten, wenn der Platz auf dem Heap bereits erschöpft ist. Die Situation kann ebenfalls auftreten, wenn bei einem Unterprogrammaufruf der benötigte Speicherplatz für lokale Variable nicht mehr vorhanden ist. Die dynamische Speicherverwaltung stellt eine der größten Schwachstellen für sicherheitskritische Systeme dar. Wenn immer möglich sollte nach der Startup- und Initialsierungsphase des Programmes auf dynamische Instanziierung verzichtet werden.

3.6. Understandabilit (Verständlich eit)

Ada setzt an die Entwickler keine hohen Kompetenzen bezüglich, der Codeverständlichkeit, (was ja eigentlich eine subjektive Meinung eines Programmierers ist). Die Sprache besitzt einfache Definitionen und liefert einen ausdruckstarken Quellcode.

3. . Testabilit (Testbar eit) Verifiabilit (Verifizierbar eit)

Ada Compiler führt Extensive Checks während der Kompilierzeit durch. Dies führt dazu, dass die normalen Flüchtigkeitsfehler die bei anderen Programmiersprachen zu extremem Testaufwand führen, bei Ada schon während der Kompilierzeit entdeckt werden.
Weiterhin wird die Software-Revision durch einige Annexe unterstützt. Das Pragma *Reviewable* veranlasst die Darstellung des erzeugten Maschinencode sowie der Abbildung der Datenobjekte. Es wird auch die Lebensdauer der Objekte, die Art und Umfang der generierten Laufzeitüberprüfungen und der Umfang des Laufzeitsystems selbst analysiert. Das Pragma *Inspection_Point* legt die Wertzuordnung der betroffenen Objekte offen.

3. . Recoverabilit (iederherstellbar eit)

Ada besitzt einen ausgeklügelten Mechanismus zur Auslösung und Behandlung von Laufzeitfehlern (exception handling), man spricht allgemein von Ausnahmen. Im Ada-Laufzeitsystem sind einige Ausnahmesituationen vordefiniert, die alle die möglichen Laufzeitfehler abdecken, die sich rein aus der Semantik der Sprachdefinition ergeben.
Eine Ausnahmenbehandlung ist für alle Ablaufstrukturen, die in Ada vorgesehen sind, möglich und sie bezieht sich auf die jeweilige Prozedur, Funktion oder einen Anweisungsblock. Im Zusammenhang mit der Ausführung spricht man allgemein auch von Rahmen, die während der Ausführung, nach dem LIFO-Prinzip, eine Kette bilden. Tritt der Kontrollfluss in einen Block ein, oder es wird eine Prozedur aufgerufen, so wird ein neuer, untergeordneter Rahmen erzeugt. Dieser wird beim verlassen der Prozedur oder des Blocks wieder entfernt.
Falls nun eine Ausnahmesituation im einem grade bearbeitenden Rahmen eintritt, und ist im betreffenden Ausnahmebehandlungsteil eine entsprechende Ausnahmesituation mit einer Anweisungsfolge angeführt, so geht der Kontrollfluss auf diese Anweisungsfolge über. Die Ausnahme ist damit beendet. Danach wird der Rahmen, der die Ausnahmenbehandlung enthält, auf jeden Fall abgeschlossen und die Programmausführung wird mit dem übergeordneten Rahmen fortgesetzt.

3. .1. Arithmetische Laufzeit ber achung (operational arithmetic)

Die arithmetische Laufzeitüberwachung übernimmt ebenfalls der Ausnahmenmechanismus. In Ada 3 gab es noch eine spezielle Ausnahme *Numeric*_Error, die jedoch bei neueren Implementierungen der Sprache, auf jeden Fall jedoch in Ada 5, von *Constraint_Error* voll abgedeckt wird.

3. . Maintainabilit (artbar eit) Extendabilit (Er eiterbar eit) und iederver endbar eit

Das im Ada benutzte Package Konzept bietet die Möglichkeit große Softwareprojekte zu gliedern und zu Modularisieren.

- Die separate Compilierbarkeit von Ada-Units, also Unterprogrammen, erleichtert die Wartung und fordert die Wiederverwendbarkeit. Weiterhin kennt Ada *generische Einheiten* die nochmals die Wiederverwendbarkeit der Ada Unterprogramme steigert. Diese Einheiten sind eigentlich Programm-Schablonen für welche eine Parameter vereinbart worden sind.

3.10. sicheres Subset

Ada besitzt eine sicheres Subset SPARK.
SPARK ist eine Teilsprache von Ada, zum Compilieren von SPARK werden sogar Ada-Compiler benutzt, aber SPARK ist eine eigenständige Ada-ähnliche Programmiersprache. Der Hauptunterschied ist der, dass viele Konzepte von Ada fehlen. Der Sprachkern ist gegenüber den Standard Ada Sprachkern stark eingeschränkt.
SPAK verzichtet auf:

- Ausnahmen-Behandlung
- Generische Einheiten
- Zugriffstypen (Zeiger)
- Dynamische Speicherverwaltung
- Rekursive Unterprogramme

4. usammenfassung

4.1. Vor ort

Es hört sich eigentlich ganz einfach eine Programmiersprache die für den Einsatz in SkS empfohlen wird, nach der ihrer Eignung für diesen Einsatz zu untersuchen. Leider stellte sich die Aufgabe doch etwas schwieriger dar als ich mir erhofft habe, sei es wegen der vielen widersprüchlichen Angaben in der Literatur oder durch das umfangreiche Sicherheitskonzept von Ada.
Nichts desto trotz, hoffe ich eine objektive und vor allem technisch korrekte Beurteilung getroffen zu haben.

4.2. eurteilung von Ada

Im laufe der Jahre die Ada am Markt verfügbar ist, sind eine Menge Werkzeuge, teilweise komplette Entwicklungsumgebungen, für Ada entwickelt und programmiert worden.
Ada Compiler werden streng nach dem Ada Standard validiert, es sollten nur solche Compiler eingesetzt werden die den Validierungsprozess erfolgreich abgeschlossen haben. Diese Compiler haben sich durch den tausendfachen Einsatz in der Praxis als sehr zuverlässig erwiesen.

Da fundierte Ada – Kenntnisse in der Softwarebranche, die den Raum- und Luftfahrt sowie den militärischen Sektor beliefert, als Vorraussetzung für die Erteilung von Aufträgen gelten, ist das Interesse nach Ada Produkten wie Werkzeugen aber auch nach Expertenwissen groß.
Viele Unternehmen, vor allem Compilerhersteller, bieten Ada Schulungen wo man sich das Wissen aneignen kann, außerdem ist Ada oft in der Literatur beschrieben worden.

Ada ist als eine Universalsprache entwickelt worden, die ein breites Spektrum von Anwendungsszenarien abdeckt. Aus diesem Grund ist die Sprache sehr voluminös, es ist eine Vielzahl von Konstrukten in den Sprachkern aufgenommen worden, die sich nicht ganz so gut für den Einsatz in sicherheitskritischen Systemen eignen. Viele dieser Punkte tauchen in dem Anforderungskatalog, den ich im Kapitel 2.5 aufgestellt habe, und sind im Kapitel 3 schon beschrieben und beurteilt worden. In einigen Punkten erkennt man, dass die Sicherheitsmassnahmen die Ada standardmäßig mit sich bringt nicht ausreichend sind. Die Entwickler von Ada Haben dies auch erkannt, und haben deshalb den, schon teilweise beschriebenen, Annex Sicherheit konzipiert durch welchen u.a. eine ganze Reihe von Sprachkonstrukten von der Benutzung ausgeschlossen werden kann. Diese *Restrictions* betreffen folgende Bereiche :

- Prozesse und geschützte Typen (Einschränkung oder Verbot)
- Verwaltung des Speichers (Einschränkung oder Verbot)
- Verbot der Ausnahmebehandlung
- Einschränkung der verwendbaren Datentypen (z.B. keine Gleitkommazahlen)
- Einschränkung der Ein-/Ausgabe
- Verbot der Polymorphie
- Verbot der Rekursion
- Verbot reentranter Programmteile

Mit dem flexiblen Einsatz von Spracheinschränkungen eignet sich Ada für den Einsatz in SkS, wobei ein leichtsinniger Umgang damit auch fatale Folgen haben kann. Eine Ariane 5 Rakete ist abgestürzt, weil aus Performance Gründen die Ausnahmebehandlung eingeschränkt worden ist, und so ein Softwarefehler nicht abgefangen werden konnte.

Dies zeigt uns eigentlich, dass auch sichere Programmiersprachen, unsichere Programme produzieren können, und dass letztendlich der Entwickler für die Sicherheit verantwortlich ist.

5. Anhang

5.1. Literaturverzeichnis

[Sto 6] Neil Storey. *Safety Critical Computer Systems.* Addison-Wesley, 1 6.

[Gol 5] S.J. Goldsack. *Ada for specifications: possibilities and limitations,* Cambridge University Press, 1 5.

[Kra 6] K.P. Kratzer, *Ada Eine Einführung für Programmierer,* Hanser, 1 6.

[Gol 2] S.J. Goldsack. *Programming Embedded Systems with ADA,* Prentice/Hall, 1 2.

[Gie03] Giese, H.: Folien zur Vorlesung Software Engineering for Safety Critical Computer Systems. Universität Paderborn, WS 02/03

[Gie04] Giese, H.: Folien zur Vorlesung Software Engineering for Safety Critical Computer Systems. Universität Paderborn, WS03/04

[Loe04] Löper, C.: Anforderungsanalyse und Anforderungsdefinition für sicherheitskritische Systeme. Universität Paderborn, WS03/04

[Carr 0] B. A. Carr , T. J. Jennings, F. J. Maclennan, P. F. Farrow, and J. R. Garnsworthy, SPARK – The SPADE Ada Kernel, 3rd ed, Program Validation Limited, 1 0.

[Ada 5] GNAT Ada 5 Reference Manual

Verifikation, Validation und Testen von Sicherheitskritischen Systeme

Seminararbeit
im Fach Informatik
im Rahmen des Seminars "**Sicherheitskritische Systeme**"
an der

Universität Siegen, Fachgruppe für Praktische Informatik

eingereicht bei
Dr. Jörg Niere

vorgelegt von
Chandra Kurnia Jaya

Sommersemester 2004

Inhaltsverzeichnis

1 Beispiele für Fehler in Software..... 3

2 Verifikation, Validierung und Testen 4

3 Black-Box-Test..... 5

3.1 Äquivalenzklassenbildung..... 6

3.2 Grenzwertanalyse 10

3.3 Test spezieller Werte Grenzwertanalyse 13

3.4 Ursache-Wirkungs-Graph..... 14

4.White-Box-Test .....…….. 18

4.1 Beweis durch Widersprüche..... 18

5. Black-Box-Test gegen White-Box-Test…… 21

6. Testprinzipien..... 22

7. Zusammenfassung..... 24

8. Literaturverzeichnis..... 25

1. Beispiele für Fehler in Software

Im Bereich der Medizin gibt es auf Grund von Softwarefehlern viele Todesfälle. Mehrere Patienten starben, nachdem sie wegen Krebs mit Therac-25 bestrahlt wurden. Wegen einer zu hohen Strahlendosis wurden ihre Zellen nachhaltig geschädigt. Das medizinische Gerät Therac-25 ist ein linearer Teilchenbeschleuniger zur Strahlentherapie für die krebskranken Patienten. Insgesamt sind 11 Therac-25 Geräte in Kanada und USA installiert. Das Bild zeigt die Software von Therac-25.

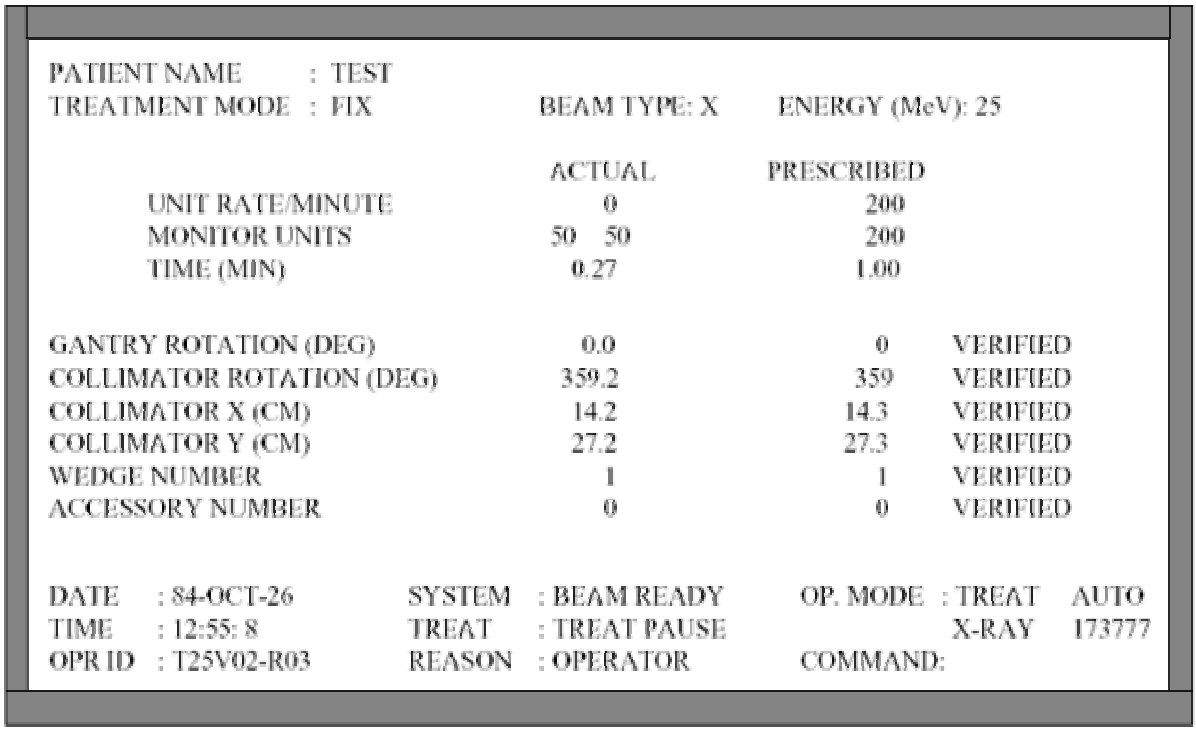


Abbildung 1.1 : Darstellung des Benutzer-Interfaces

Bei Fehlfunktionen wurden Fehlermeldungen auf dem Bildschirm dargestellt. Der häufigste Fehler war „Malfunction 54“. Diese Meldung ist sehr kryptisch und in der Dokumentation wurde „Malfunction 54“ kurz als dose input 2 beschrieben. Da Fehler sehr oft auftraten, wurden diese Fehler als nicht schlimm von den Operatoren betrachtet. Die Operatoren setzten damit die Behandlung fort. Sie wussten nicht, dass die nichts sagende Fehlermeldung „Malfunction 54“ bedeutete, dass die Strahlendosis, mit der die Patienten bestrahlt wurden entweder zu hoch oder zu niedrig war. Als schließlich das Therac-25 System näher untersucht wurde, stellte es sich heraus, dass die Schwachstellen und die Fehler an der Software lagen.

Fazit :

Die Software der Maschine vor allem die kritischen Komponenten müssen ausreichend getestet werden und die Fehlermeldungen müssen verständlich und lesbar dokumentiert werden.

2. Verifikation, Validierung und Testen

In diesem Seminar wird das Thema Verifikation, Validierung und Testen von sicherheitskritischen Systemen behandelt. Diese Ausarbeitung wird sich vor allem auf das Testen von sicherheitskritischer Software konzentrieren.

Definition 2.1 (Die sicherheitskritische Software) *Die sicherheiskritische Software ist Software, deren Ausfall eine Auswirkung auf die Sicherheit haben könnte oder großen finanziellen Verlust oder sozialen Verlust verursachen könnte [9].*

Definition 2.2 (Verifikation) *Verifikation ist der Prozess des Überprüfens, der sicherstellt, dass die Ausgabe einer Lebenszyklusphase die durch die vorhergehende Phase spezifizierten Anforderungen erfüllt [9].*

Definition 2.3 (Validation) *Validation ist der Prozess des Bestätigens, dass die Spezifikation einer Phase oder des Gesamtsystems passend zu und konsistent mit den Anforderungen des Kunden ist [9].*

Nach der Definition sind Verifikation und Validierung nicht dasselbe. Der Unterschied zwischen ihnen wurde nach Boehm (1979) so ausgedrückt:
„Validierung : Erstellen wir das richtige Produkt ?“
„Verifikation : Erstellen wir das Produkt richtig ?“

Definition 2.4 (Spezifikation) *Spezifikation ist ein Test, der die Syntax und Semantik eines bestimmten Bestandteiles beschreibt bzw. eine deklarative Beschreibung, was etwas ist oder tut [9].*

Definition 2.5 (Testen) *Das Testen ist der Prozess, ein Programm auf systematische Art und Weise auszuführen, um Fehler zu finden [10].*

Während die Verifikation den Output einer Entwicklungsphase auf die Korrektheit mit der vorherigen Phase untersucht, wird die Validation benutzt, um das Gesamtsystem mit den Kundenanforderungen zu vergleichen.
Die zentrale Tätigkeit bei Validation ist das Testen. Das Gesamtsystem wird bei Ende des Prozess getestet, ob es den Kundenanforderungen entspricht oder nicht. Ein eigenes Kapitel ist dem Testen gewidmet, deswegen wird es an dieser Stelle nicht erklärt. Die zentrale Tätigkeit bei der Verifikation ist der Beweis mit der formalen Verifikation. Dieser Beweis wird nicht in dieser Ausarbeitung behandelt.

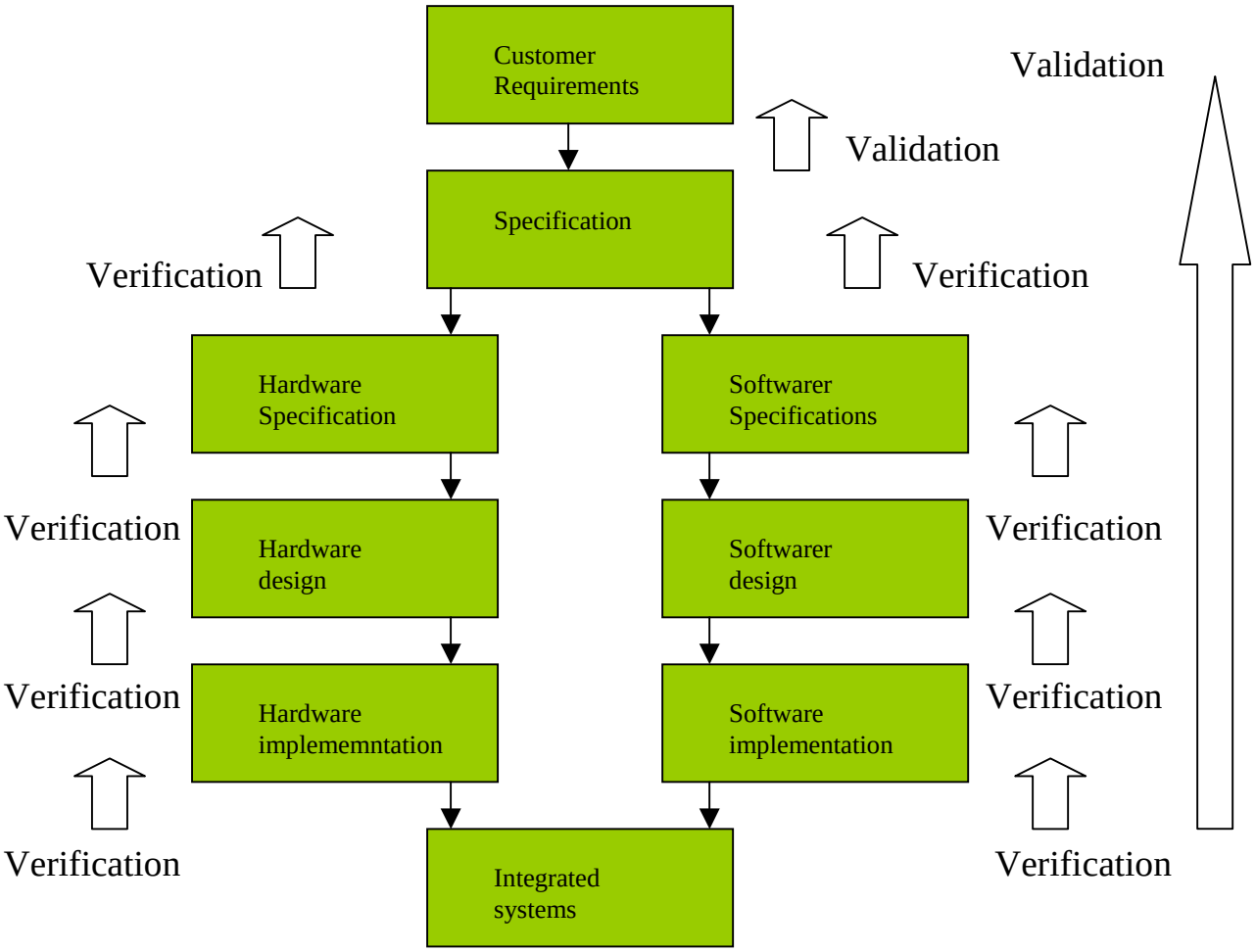
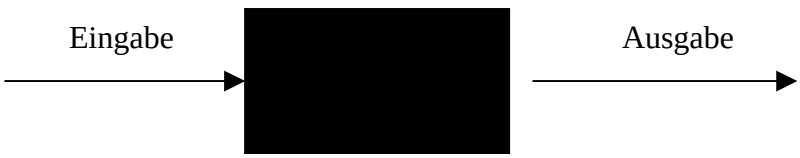


Abbildung 2.1 : Entwicklungsphase mit dem formalen Verifikationsnprozess

Es gibt zwei grundlegende Strategie für das Testen von Software. Die erste Strategie wird Black-Box-Test genannt. Black-Box-Test bedeutet, dass der Tester nicht die Kenntnisse über das interne Verhalten und die Struktur des Programms hat.
Die zweite Strategie wird White-Box-Test genannt. Um diese Test auszuführen, muss der Tester über Kenntnisse der inneren Struktur des Programms verfügen.
Die beiden Teststrategien werden hier ausführlich erklärt.

3. Black-Box-Test



Der Tester kennt beim Black-Box-Test nur was eine Funktion macht aber nicht wie diese Funktion arbeitet. Der Tester muss nicht Programmierkenntnisse haben und er orientiert sich nur am Benutzerhandbuch, Lastheft der Software, Spezifikation der Software, um die Testfälle als Eingabe zu definieren. Die Ausgabe wird danach verglichen, ob sie gleich ist mit der richtigen Ausgabe, die in der Spezifikation steht.
Um alle Fehler zu finden, muss ein vollständiger Test ausgeführt werden. Das bedeutet, dass nicht nur alle zulässigen Eingaben getestet werden müssen, sondern die fehlerhaften Eingaben müssen auch getestet werden.

Für die Testfallbestimmung gibt es drei wichtige Verfahren:
* Äquivalenzklassenbildung (Equivalence Partitioning)
* Grenzwertanalyse (Boundary Value Analysis)
* Test spezieller Werte (Error-Guessing)

3.1 Äquivalenzklassenbildung (Equivalence Partitioning)

Eine Äquivalenzklasse ist eine Menge von Eingabewerten, die auf ein Programm eine gleichartige Wirkung ausüben. Das bedeutet, dass wenn ein Element in einer Äquivalenzklasse als Eingabe zu einem Fehler führt, alle anderen Elemente in dieser Klasse mit der größten Wahrscheinlichkeit zu dem gleichen Fehler führen werden.
Wenn ein Testfall in einer Äquivalenzklasse keinen Fehler entdeckt, so erwartet man, dass alle anderen Testfälle keine Fehler entdecken.

Wir betrachten das erste Beispiel
Das Testprogramm sieht so aus:

```
/* COPYRIGHT © 1994 by George E. Thaller
   All rights reserved
   Function : Black Box
   Equivalence Partitioning
*/


   #include <stdio.h>
   main ()
   {
            int i, z, day, month, year ;
            printf("\nTAG   MONAT   WOCHENTAG\n\n");

            /* Test Case 1 */
            day = 22; month = 6; year = 1994;
```

```
        z = week_d(day, month, year);
        printf ("%2d    %2d     %1d\n", day, month, z);

        /* Test Case 2 */
        day = 19; month = 5; year = 1994;
        z = week_d(day, month, year);
        printf("%2d     %2d      %1d\n", day, month, z);
}
```

Das Modul wird mit der folgenden Anweisung aufgerufen:
week_d (day, month, year);
Wir interessieren uns nicht im Sinne eines Black-Box-Tests, wie der Quellcode dieses Moduls aussieht und wie das Ergebnis berechnet wird.
Das Datum eines Tages wird als Parameter gegeben und das Ergebnis ist eine Zahl als Wochentag: der Sonntag bekommt die Zahl 0, der Montag bekommt die Zahl 1 und dann so fort bis Samstag. Wenn dieses Testprogramm ausgeführt wird, werden die folgenden Ergebnisse geliefert:

TAG	MONAT	WOCHENTAG
22	6	3
19	5	4

Wenn wir in den Kalender ansehen, wissen wir, dass die beiden Werte in Ordnung sind.

Es werden eine gültige Äquivalenzklasse und zwei ungültige Äquivalenzklassen für den Monat im obigen Beispiel gebildet.

Eine gültige Äquivalenzklasse :
$1 <= m <= 12$
Zwei ungültige Äquivalenzklassen :
$m < 1$ und $m > 12$

Es werden eine gültige Äquivalenzklasse und zwei ungültige Äquivalenzklassen für den Tag gebildet.

Eine gültige Äquivalenzklasse :
$1 <= t <= 31$
Zwei ungültige Äquivalenzklasse :
$t < 1$ und $t > 31$

Beim Jahr haben wir - aus der Sicht des Black-Box-Tests – erst keine Kriterien, um ein bestimmtes Jahr auszuwählen.

Wir betrachten nun das zweite Beispiel:

Spezifikation zur Ableitung des technischen Eintrittsalters einer Person in einen Versicherungsvertrag.

Eingabe : vertragsbeginn, geburtsdatum	
Hilsvariable : diff_Monat := Monat (vertragsbeginn) – Monat (geburtsdatum) diff_Jahr := Jahr (vertragsbeginn) – Jahr (geburtsdatum)	
technisches_Eintrittsalter	**Bedingung**
Fehler	vertragsbeginn < geburtsdatum
diff_Jahr	vertragsbeginn >= geburtsdatum und -5 <= diff_Monat <= 6
diff_Jahr + 1	vertragsbeginn >= geburtsdatum und diff_Monat > 6
diff_Jahr - 1	vertragsbeginn >= geburtsdatum und diff_Monat < -5

Testfall			Ausgewähltes Testdatum		
	Äquivalenzklasse	Ausgabe	Geburts-datum	Vertrags-beginn	Ausgabe :soll
T1	1 Vertragsbeginn vor Geburtsdatum	Fehler	01.02.2001	01.01.2001	Fehler
T2	2 diff_Monat im Interval [-5, 6]	diff_Jahr	01.06.1975	01.08.2001	26
T3	3 diff_Monat > 6	diff_Jahr+1	01.05.1975	01.12.2001	27
T4	4 diff_Monat < -5	diff_Jahr-1	01.10.1975	01.01.2001	25

Klasse 1 ist eine Klasse ungültiger Werte.
Weitere ungültige Klassen sind Tag, Monat, Jahr, die nicht im gültigen Wertebereich liegt (Zum Beispiel : negative Zahlen).

Auffinden der Äquivalenzklasse :

1. Wenn der Eingabewert als Wertebereich spezifiziert wird (zum Beispiel die Variable kann einen Wert zwischen 1 und 12 annehmen), so bildet man eine gültige Äquivalenzklasse (1 <= x <= 12) und zwei ungültigen Äquivalenzklassen (x <= 1 und x >12).

2. Wenn der Eingabewert als Anzahl der Werte spezifiziert wird (zum Beispiel es können bis zu vier Besitzer für ein Haus registriert sein), so bildet man eine Äquivalenzklasse mit gültigen Werten und zwei Klassen mit ungültigen Werten (kein Besitzer und mehr als vier Besitzer).

3. Wenn die Eingabebedingung eine Situation mit „... muss sein...“ verlangt (zum Beispiel das erste Zeichen des Merkmals *muss* ein Buchstabe *sein*), so bildet man eine gültige Äquivalenzklasse (es ist ein Buchstabe) und eine ungültige Äquivalenzklasse (es ist kein Buchstabe).

4. Falls die Eingangbedingung als Menge vom gültigen Eingabewert spezifiziert wird, so bildet man eine gültige Äquivalenzklasse, die aus allen Elementen der Menge besteht und eine ungültige Äquivalenzklasse außerhalb dieser gültigen Menge.

5. Wenn man vermutet, dass die Elemente in einer Äquivalenzklasse vom Programm nicht gleichwertig behandelt werden, so spalte man die Äquivalenzklasse in kleinere Äquivalenzklassen auf.

Vorteile der Äquivalenzklassenbildung :

1. Äquivalenzklassenbildung ist die Basis für die Grenzwertanalyse.

2. Äquivalenzklassenbildung ist ein geeignetes Verfahren, um aus Spezifikationen repräsentative Testfälle abzuleiten.

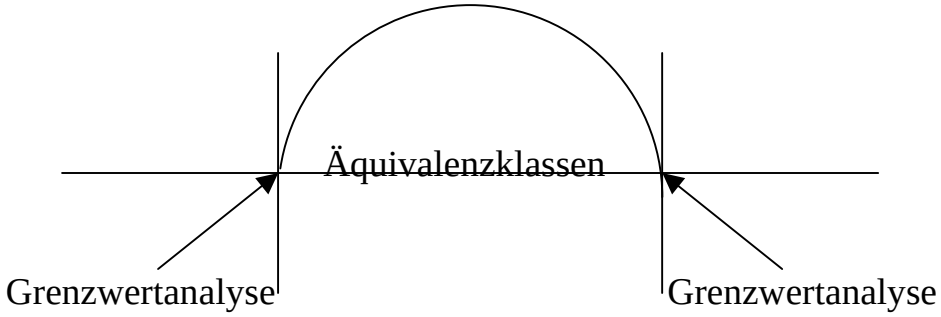
Nachteile von der Äquivalenzklassenbildung :

1. Es werden nur einzelne Eingaben betrachtet. Beziehungen, Wechselwirkungen zwischen Werten werden nicht behandelt.

3.2 Grenzwertanalyse (Boundary Value Analysis)

Grenzwertanalyse ist auf der Äquivalenzklassenbildung basierend, denn Grenzwertanalyse benutzt Testdaten von Äquivalenzklassen, welche nur die Werte an den Rändern berücksichtigt. Erfahrungen haben gezeigt, dass viele Fehler in der Nähe von Rändern stecken, deswegen wird dieses Verfahren sehr oft benutzt, um die Fehler an den Rändern zu entdecken.

Das Bild zeigt den Unterschied zwischen Grenzwertanalyse und Äquivalenzklassenbildung.



Wir betrachten das erste Beispiel

```
 /* COPYRIGHT © 1994 by George E. Thaller
    All rights reserved
    Function : Black Box
    Grenzwerte Testprogramm
*/
#include <stdio.h>
main ()
{
        inti, n, wd, day, month, year;


        printf("\n TESTFALL   TAG   MONAT   JAHR   WOCHENTAG");

        year =1994;
        n = 11;

        for ( i = 3; i <= n ; i++)
        {
                if (i = = 3)   { day = 1 ; month = 1; }
                if (i = = 4)   { day = 31; month = 12; }
                if (i = = 5)   { day = 0 ; month = 1; }

                if (i = = 6)   { day = 32 ; month ; }
```

```
                if (i = = 7)   { day = 7 ; month = 0; }
                if (i = = 8)   { day = 7 ; month = 13; }

                wd = week_d( day, month, year);
                printf (“     %2d     %2d     %2d      %4d
                %ld\n”,I, day, month, year, wd);
        }
}
```

Im Sinne eines Black-Box-Tests interessiert uns nicht, was das Modul week_d mit den Werten macht und wie das Ergebnis zustande kommt.
Wir sehen nun das Ergebnis:

TESTFALL	TAG	MONAT	JAHR	WOCHENTAG
3	1	1	1994	6
4	31	12	1994	6
5	0	1	1994	5
6	32	1	1994	2
7	7	0	1994	2
8	7	13	1994	6

Wenn wir das Ergebnis mit dem Kalender vergleichen, stellen wir fest, dass die Eingabewerte ab Testfall 5 sind falsch. Wir überprüfen hier zunächst die Grenzwerte vom Tag (0, 1, 31 und 32) und es hat sich herausgestellt, dass der Eingabewerte 0 und 32 das falsche Ergebnis liefert. Danach wird der Grenzwert für den Monat untersucht. Der Eingabewert 0 und 13 liefert das falsche Ergebnis.
Der Ausdruck „Grenzwert“ bedeutet, dass die benachbarten Werte auch berücksichtigt werden sollen. Beim Monat werden zum Beispiel nicht nur die Werte 1 und 12 sondern auch die benachbarten Werte 0, 2, 11, 13 untersucht. Warum ist das so?

Der Grund liegt darin, dass der Fehler leicht beim Eintippen des Quellcodes auftreten kann. Zum Beispiel:

< 12 obwohl man eigentlich < = 12 gemeint hat.

Dieser Flüchtigkeitsfehler ist sehr schwierig zu entdecken. Wenn man den Grenzwert untersucht, wird ein derartiger Fehler gefunden.

Wir betrachten nun das zweite Beispiel. Dieses Beispiel ist identisch mit dem Beispiel von der Äquivalenzklasse und wir erweitern und untersuchen die Testfälle um die Grenzwerte.

Spezifikation zur Ableitung des technischen Eintrittsalters einer Person in einen Versicherungsvertrag.

Eingabe : vertragsbeginn, geburtsdatum	
Hilsvariable : diff_Monat := Monat (vertragsbeginn) – Monat (geburtsdatum) diff_Jahr := Jahr (vertragsbeginn) – Jahr (geburtsdatum)	
technisches_Eintrittsalter	**Bedingung**
Fehler	vertragsbeginn < geburtsdatum
diff_Jahr	vertragsbeginn >= geburtsdatum und -5 <= diff_Monat <= 6
diff_Jahr + 1	vertragsbeginn >= geburtsdatum und diff_Monat > 6
diff_Jahr - 1	vertragsbeginn >= geburtsdatum und diff_Monat < -5

Ti-j : Testfall in Äquivalenzklasse i an der Grenze zu Klasse j

Testfall			Ausgewähltes Testdatum		
	Eingabe	Ausgabe	Geburts-datum	Vertrags-beginn	Ausgabe :soll
T1-2	Vertragsbeginn 1 Tag vor Geburtsdatum	Fehler	02.02.2001	01.02.2001	Fehler
T2-1	Vertragsbeginn = Geburtsdatum	0	01.06.1975	01.06.1975	0
T2-3	diff_Monat = 6	diff_Jahr	01.06.1975	01.12.2001	26
T2-4	diff_Monat = -5	diff_Jahr	01.06.1975	01.01.2001	26
T3-2	diff_Monat = 7	diff_Jahr+1	01.05.1975	01.12.2001	27
T4-2	diff_Monat = -6	diff_Jahr-1	01.07.1975	01.01.2001	25

Wir können leicht verstehen, dass wir hier diff_Monat = -5, -6, 6, 7 für den Grenzwert benutzen. Außerdem wird hier untersucht, wie sich das Programm bei falschen Werten (hier Vertragsbeginn 1 Tag vor Geburtsdatum) verhält.

Einige Richtlinien bei Bildung des Grenzwertes:

1. Wenn ein Wertebereich für die Eingabebedingung in der Spezifikation steht, so muss man Testfälle für den unteren und oberen Grenzwert

entwerfen, die direkt neben den Grenzwert liegen. Zum Beispiel: die Eingabewerte liegen
zwischen -5 <= x <= 6, so entwirft man Testfälle für die Situation mit -5, -6 und 6, 7.

2. Wenn die Ein- oder Ausgabe eines Programms eine geordnete Menge (zum Beispiel: lineare Liste oder Tabelle) ist, muss man Testfälle, die aus dem ersten und letzten Elemente der Menge bestehen, konstruieren.

3. Wenn die Eingabebedingung als Anzahl der Werte spezifiziert wird, muss man sowohl das Maximum und das Minimum als die gültigen Eingabewerte als auch ein weniger und ein hoher als das Minimum und das Maximum als die ungültigen Eingabewerte entwerfen.
   Zum Beispiel: es können von ein bis zu vier Besitzer für ein Haus registriert wird. Man soll Testfälle für 0,1 Besitzer und 4, 5 Besitzer entwerfen.

Vorteile der Grenzwertanalyse :

1. Grenzwertanalyse verbessert die Äquivalenzklassenbildung, denn die Grenzwertanalyse untersucht die Werte an den Grenzen der Äquivalenzklassen. Wir wissen schon, dass Fehler häufiger an den Grenzen von Äquivalenzklassen zu finden sind, als innerhalb dieser Klassen.

2. Grenzwertanalyse ist bei richtiger Anwendung eine der nützlichsten Methoden für den Testfallentwurf.

Nachteile der Grenzwertanalyse :

1. Es ist schwierig, Rezepte für die Grenzwertanalyse zu geben, denn dieses Verfahren erfordert die Kreativität vom Tester für das gegebene Problem.

2. Es ist schwierig alle relevanten Grenzwerte zu bestimmen.

3. Es werden nur einzelne Eingaben betrachtet. Beziehungen und Wechselwirkungen zwischen Werten werden nicht behandelt.

3.3 Test spezieller Werte (Error-Guessing)

Das Error-Guessing ist im eigentlichen Sinne keine Testmethode, sondern dient zur Erweiterung und Optimierung von Testfällen.
Diese Methode beruht auf der Erfahrung und dem Wissen des Testers und muss nicht von der Spezifikation abgeleitet werden. Aufgrund seiner langjährigen

Tätigkeit als Tester oder Programmierer kennt dieser zum Beispiel die häufig aufgetretenen Fehler. Bei dieser Methode ist es schwierig die Vorgehensweise anzugeben, da es ein intuitiver Prozess ist. Prinzipiell legt man eine Liste möglicher Fehler oder fehleranfälliger Situationen an und definiert damit die neuen Testfälle.

Beispiele für Error-Guessing :

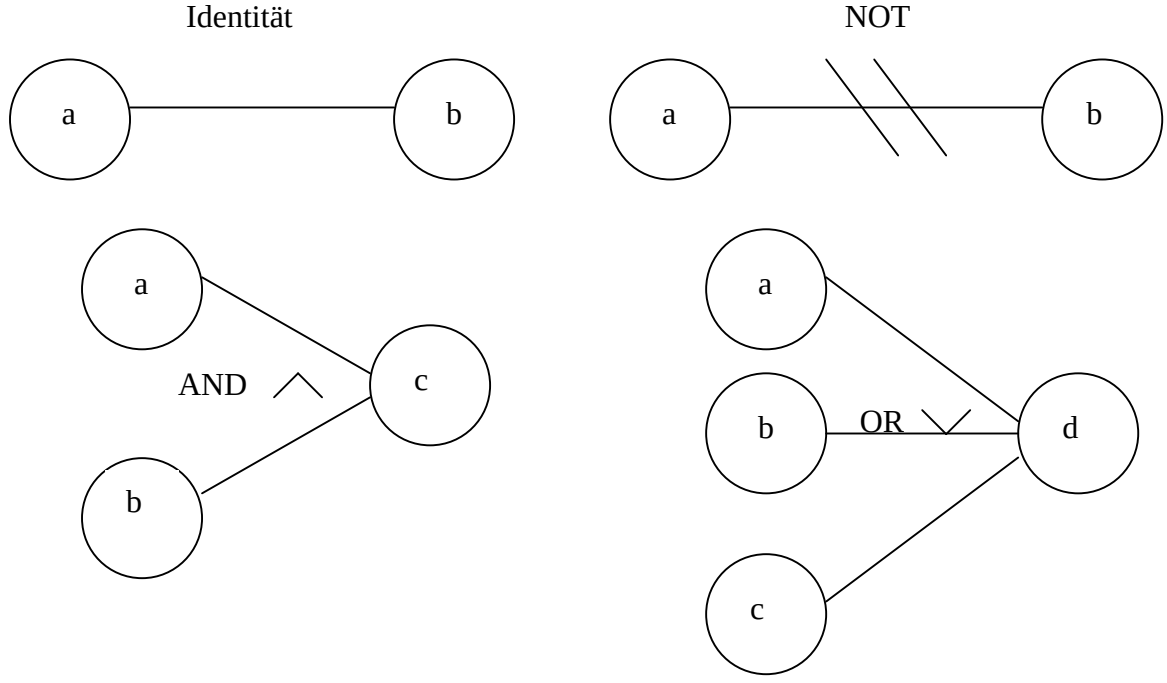
1. Der Wert 0 als Eingabewert zeigt oft eine fehleranfällige Situation.

2. Bei der Eingabe von Zeichenketten sind Sonderzeichen besonders sorgfältig zu betrachten.

3. Bei der Tabellenverarbeitung stellen kein Eintrag und ein Eintrag oft Sonderfälle dar.

3.4 Ursache-Wirkungs-Graph (Verbesserung von Äquivalenzklassen und Grenzwertanalyse)

Äquivalenzklassenbildung und Grenzwertanalyse haben eine gemeinsame Schwäche: sie können keine Kombination von Eingabebedingung überprüfen. Diese Schwäche wird durch den Ursache-Wirkungs-Graph behoben.
Ein Ursache-Wirkungs-Graph ist eine formale Sprache, in die eine Spezifikation aus der natürlichen Sprache übersetzt wird. Der Graph entspricht einer Schaltung der Digitallogik. Es sind dabei nur die Operatoren aus der Booleschen Algebra als Vorkenntnisse notwendig.
Beispiel für die Notation eines Ursache-Wirkungs-Graphs :



Jeder Knoten kann den Wert 0 oder 1 annehmen.
- Die Funktion der Identität besagt, wenn a = 1 ist, dann b= 1 ist, ansonsten b= 0.
- Die NOT-Funktion besagt, wenn a = 1 ist, dann b = 0 ist, ansonsten b = 1.
- Die AND-Funktion besagt, wenn a = 1 und b = 1 sind, dann c = 1 ist, ansonsten c = 0.
- Die OR-Funktion besagt, wenn a oder b oder c = 1 ist, dann d = 1, ansonsten d = 0.

Die Testfälle werden wie folgt entwickelt:

1. Der Tester muss die komplexe Spezifikation der Software in kleinere Stücke zerlegen. Zum Beispiel: ein Programm in einzelne Methoden.

2. Der Tester muss Ursachen und Wirkungen der Spezifikation festlegen. Eine Ursache ist eine Eingangsbedingung oder eine Äquivalenzklasse von Eingangsbedingungen. Eine Wirkung ist eine Ausgangsbedingung oder eine Systemtransformation (eine Nachwirkung, die die Eingabe auf den Zustand des Programms oder Systems hat). Ursachen und Wirkungen werden definiert, indem man die Spezifikation Wort für Wort liest und alle Worte, die Ursachen und Wirkungen beschreiben, unterstreicht.

3. Der semantische Inhalt der Spezifikation wird analysiert und in einen booleschen Graphen transformiert. Die Ursache wird in die linke Seite des Graphs geschrieben und die Wirkung in die rechte Seite des Graphen. Man verbindet danach Ursache und Wirkung mit den Operatoren aus der booleschen Algebra (AND, OR, NOT).

4. Der Graph wird mit Kommentaren versehen, die Kombinationen von Ursachen und/oder Wirkungen angeben, die aufgrund kontextabhängiger Beschränkungen nicht möglich sind.

5. Der Graph wird in eine Entscheidungstabelle umgesetzt. Jede Spalte stellt einen Testfall dar.

6. Die Spalten in der Entscheidungstabelle werden in die Testfälle konvertiert.

Zum besseren Verständnis eines Ursache-Wirkungs-Graphen wird hier ein Beispiel angeführt.

Angenommen, wir haben eine Spezifikation für eine Methode, die dem Benutzer erlaubt, eine Suche nach einem Buchstabe in einer vorhandenen Zeichenkette

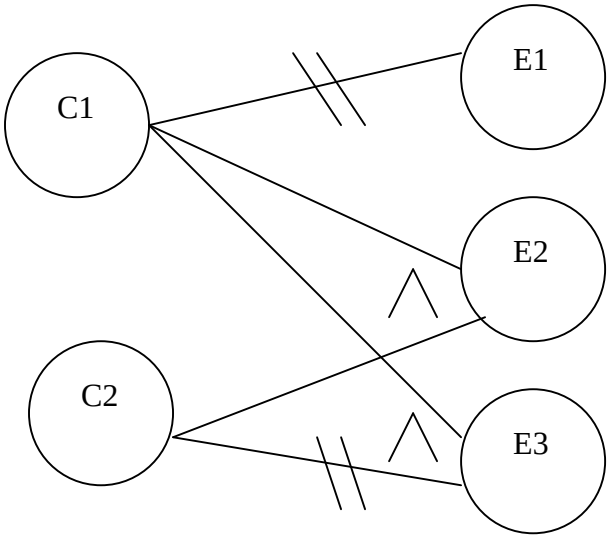
durchzuführen. Die Spezifikation besagt, dass der Benutzer die Länge von Zeichenkette (bis zu 80) und den zu suchenden Buchstabe bestimmen kann.
Wenn der gewünschte Buchstabe in der Zeichenkette erscheint, wird seine Position berichtet. Wenn der gewünschte Buchstabe nicht in der Zeichenkette erscheint, wird eine Meldung „nicht gefunden“ ausgegeben. Wenn ein Index verwendet wird, der nicht im zulässigen Bereich liegt, wird eine Fehlermeldung „out of range“ ausgegeben.

Jetzt werden die Ursachen und die Wirkungen anhand der Spezifikation festgelegt.
Die Ursachen sind:
C1 : Positive Ganzzahl von 1 bis 80
C2 : Der zu suchende Buchstabe ist in der Zeichenkette

Die Wirkungen sind:
E1 : Die Ganzzahl ist out of range
E2 : Die Position des Buchstabens in der Zeichenkette
E3 : Der Buchstabe wird in der Zeichenkette nicht gefunden

Der Verhältnis (der semantische Inhalt) wird wie folgt beschrieben:
Wenn C1 und C2, dann E2.
Wenn C1 und nicht C2, dann E3.
Wenn nicht C1, dann E1.

Im nächstens Schritt wird der Graph entwickelt. Die Ursacheknoten werden auf einem Blatt Papier links und die Wirkungsknoten rechts notiert.



Der nächste Schritt ist Entscheidungstabelle zu bilden.

Testfall	T1	T2	T§
C1	1	1	0
C2	1	0	-
E1	0	0	1
E2	1	0	0
E3	0	1	0

T1, T2, T3 sind Testfälle
E1, E2, E3 sind die Wirkung (effect)
C1, C2 sind die Ursache (cause)
0 stellt den Zustand „nicht vorhanden“, „falsch“ dar
1 stellt den Zustand „vorhanden“, „wahr“ dar
- stellt den Zustand do not care

Der Tester kann die Entscheidungstabelle benutzen, um das Programm zu testen. Zum Beispiel: Wenn die vorhandene Zeichenkette „abcde“ ist, sind die möglichen Testfälle wie folgt:

Testfälle	Länge	Der zu suchende Buchstabe	Ausgabe
T1	5	c	3
T2	5	w	Nicht gefunden
T3	90	-	Out of range

Die Vorteile von Ursache-Wirkungs-Graph :

1. Der Graph, der von der Spezifikation abgeleitet wird, erlaubt eine vollständige Kontrolle der Spezifikation.

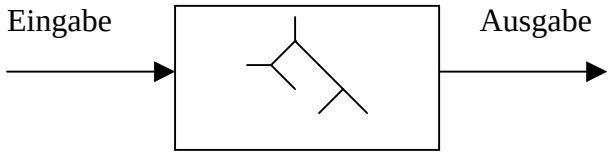
2. Unbeständigkeiten, Ungenauigkeiten werden leicht ermittelt.

3. Beziehungen und Wechselwirkungen zwischen Werten werden behandelt und die Schwäche von Äquivalenzklassenbildung und Grenzwertanalyse werden damit abgedeckt.

Die Nachteile von Ursache-Wirkungs-Graph :

Wenn die Spezifikation zu kompliziert ist, gibt es viele Ursachen und Wirkungen. Der Graph sieht kompliziert aus.

4. White-Box-Test



Der Tester muss die innere Struktur des Programms und den Quellcode kennen, weil die innere Struktur des Programms bei dieser Strategie getestet wird. Bei dieser Strategie definiert der Tester die Testdaten mit Kenntnis der Programmlogik (zum Beispiel if/else Verzweigung).
Das wichtige Prinzip beim White-Box-Test ist:

1. Jeder Programmpfad muss mindestens einmal durchlaufen werden.

2. Jeder Modul, jede Funktion muss mindestens einmal benutzt werden

Einige wichtige Verfahren von White-Box-Test :

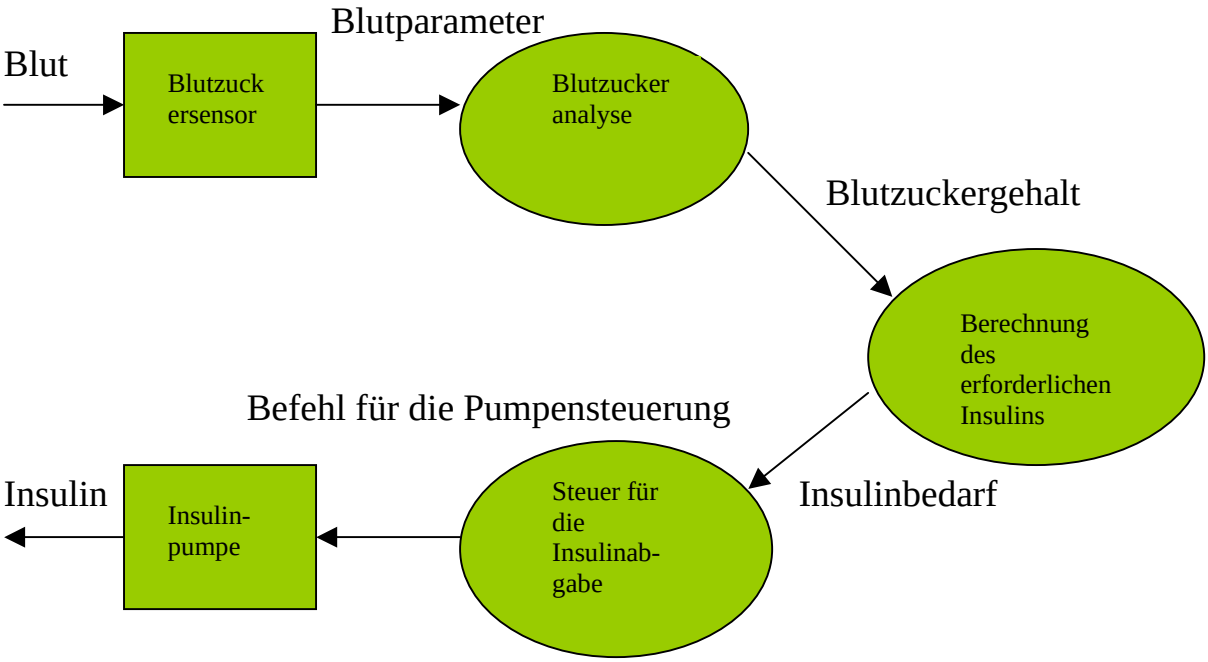
1. Beweis durch Widersprüche

2. Testdeckungsgrad (logic coverage testing)

4.1 Beweis durch Widersprüche

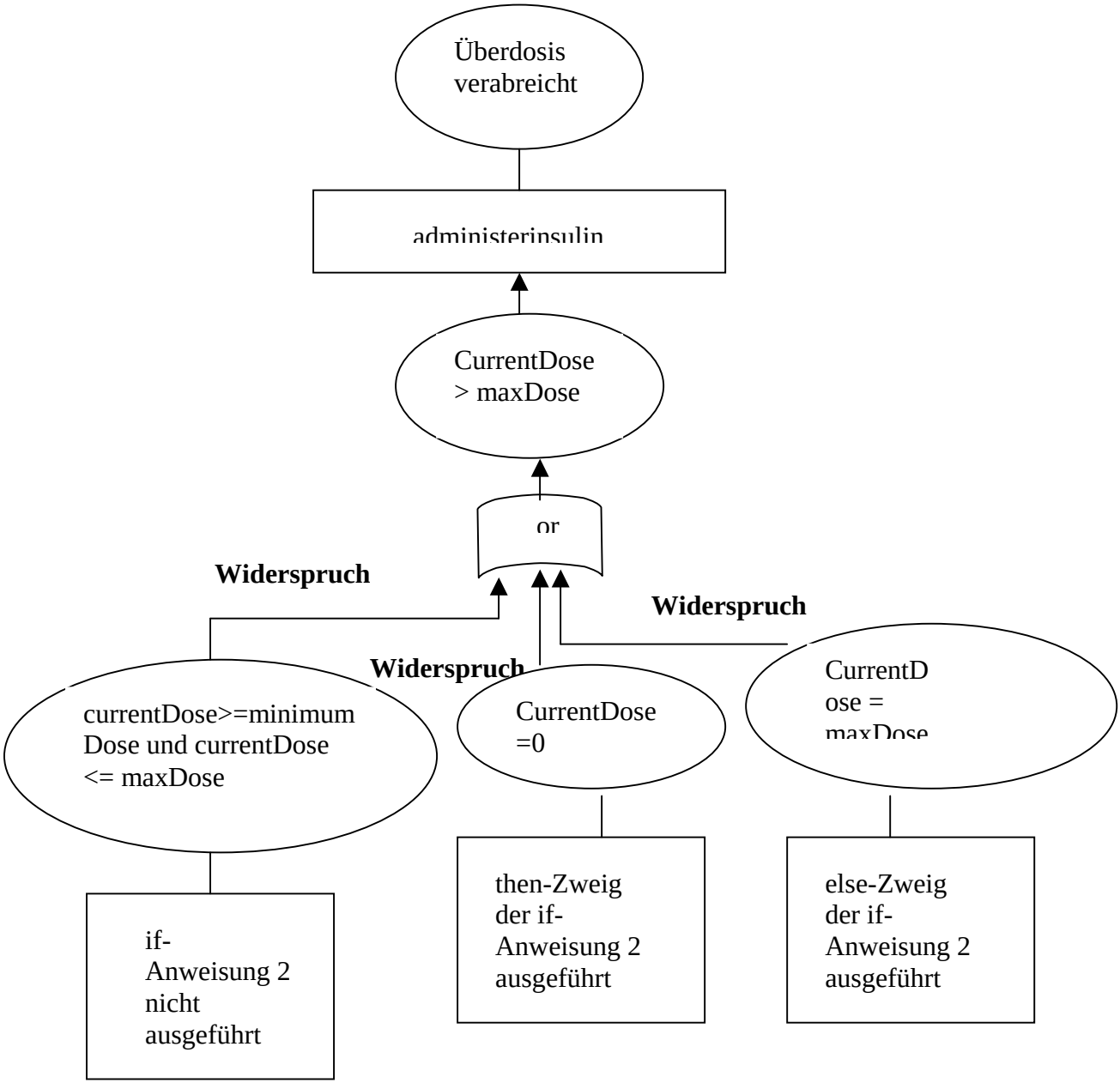
Beweis durch Widersprüche bedeutet, dass man von der Annahme ausgeht, ein unsicherer Zustand kann durch Ausführung des Programms herbeigeführt werden. Man analysiert den Code und zeigt, dass die Vorbedingungen für das Erreichen des unsicheren Zustands durch die Nachbedingungen aller Programmpfade, die zu diesem Zustand führen können, ausgeschlossen werden.

Um diese Methode zu verdeutlichen, wird hier ein einfaches sicherheitskritischens medizinisches System verwendet. Dieses System heisst Insulindosiersystem.
Insulindosiersystem ist ein Gerät, das den Blutzuckergehalt überwacht und gibt, falls erforderlich, eine angemessene Insulindosis aus.
Das Bild zeigt die Arbeitsweise eines Insulindosiersystems




```
// Quellcode für die Insulinabgabe
// COPYRIGHT © 1994 by Ian Sommerville
// Die abgegebene Insulinmenge ist eine Funktion des Blutzuckerspiegels, der zuletzt
// abgegebenen Dosis und der Zeit, zu der die letzte Dosis verabreicht wurde.

CurrentDose = computeInsulin ();

//Sicherheitsüberprüfung- Anpassung der aktuellen Dosis, wenn nötig
//if Anweisung 1

if (previousDose == 0)
{
      if (currentDose > 16)
          currentDose = 16;
}
else
     if (currentDose > (previousDose * 2) )
          currentDose = previousDose * 2;

//if Anweisung 2

if ( currentDose < minimumDose)
     currentDose = 0;
else
     if ( currentDose > maxDose)
           currentDose = maxDose;
administerInsulin (currentDose);
```

Die Vorbedingung für den unsicheren Zustand in diesem Fälle ist currentDose > maxDose. Jetzt müssen wir zeigen, dass alle Programmpfade im Widerspruch zu dieser unsicheren Annahme stehen.
Wenn dies der Fall ist, kann die unsichere Bedingung nicht wahr sein. Also ist das System sicher.
Wenn wir den Code analysieren, gibt es drei mögliche Programmpfade, die zum Aufruf der Methode administerInsulin führen :
1. Keiner der Zweige der if-Anweisung 2 wird ausgeführt. Das kann nur geschehen, wenn currentDose grösser oder gleich minimumDose und kleiner oder gleich maxDose ist.
2. Der then-Zweig von if-Anweisung 2 wird ausgeführt. In diesem Fall wird currentDose auf Null gesetzt. Seine Nachbedingung lautet currentDose = 0.
3. Der else-if Zweig der if-Anweisung 2 wird ausgeführt. In diesem Fall wird currentDose auf maxDose gesetzt. Die Nachbedingung lautet currentDose = maxDose.



In jedem der drei Fälle widersprechen die Nachbedingungen der unsicheren Vorbedingung, d.h. das System ist sicher.

5. Black-Box-Test gegen White-Box-Test

Wir haben schon gesehen, wie man mit Black-Box und White-Box Verfahren das Programm testen kann. Wir haben damit die Qualität der Software erhöht, weil einige Fehler beim Testen entdeckt werden.
Wir werden jetzt die Vorteile und die Nachteile von Black-Box-Test und White-Box-Test genauer betrachten. Welches Verfahren ist besser?

Die Vorteile von Black-Box-Test :

1. Der Tester muss die Implementierung oder den Quellcode nicht kennen

2. Die Vorgehensweise ist einfacher als White-Box-Test

3. Der Tester macht nicht denselben Fehler, wie der Implementierende.

Die Nachteile von Black-Box-Test :

1. Man weiß nicht, ob jeder Zweig durchlaufen wird.

2. Man weiß nicht, ob es unnötige Programmteile gibt.

3. Man weiß nicht, ob es kritische Programmteile gibt.

Die Vorteile von White-Box-Test :

1. Man kann sich sichern sein, dass das Programm keinen ungetesteten Code enthält.

2. Wer den Quellcode kennt weiß, wo besonders sorgfältig getestet werden muss.

Die Nachteile von White-Box-Test :

1. Die Vorgehensweise ist aufwändiger als der Black-Box-Test

2. White-Box-Test kann nicht beweisen, dass das Programm seiner Spezifikation entspricht.

6. Testprinzipien

Nachdem wir einige Methode für Testen betrachtet haben, wollen wir hier einige Testprinzipien kennen lernen. Da das Testen stark von Psychologie beeinflusst wird, werden hier einige Richtlinien, die beim Testen als Leitfaden benutzt werden sollen, erklärt.

Ein Programmierer sollte nie versuchen, sein eigenes Programm zu testen. [1]

Diese Aussage bedeutet aber nicht, dass der Programmierer nicht testen darf. Sie besagt, dass das Testen effektiver durch eine externe Gruppe ausgeführt wird.
Es gibt ein bekanntes Prinzip beim Testen: Testen ist ein destruktiver Prozess.
Der Programmierer hat sein Programm fertig gemacht, danach wird das Programm von ihm getestet. Das ist schwierig für den Programmierer, weil er die Seite wechseln muss und jetzt eine destruktive Tätigkeiten macht. Er muss eine destruktive Einstellung gegenüber seinem Programm haben.
Zusätzlich zu diesem psychologischen Problem gibt es in der Praxis eine weitere Schwierigkeit: Es kann passieren, dass der Programmierer die Spezifikation falsch verstanden hat. In diesem Fall kann der Programmierer nicht bemerken, dass es einen Widerspruch zwischen der Spezifikation und seinem Quellcode gibt.

Testfälle müssen für ungültige und unerwartete Eingabedaten ebenso wie für gültige und erwartete Eingabedaten definiert werden [1]

Diese Aussage besagt, dass der Tester nicht die ungültigen und unerwarteten Daten vernachlässigen soll. Die ungültigen und erwarteten Daten sind sehr nützlich, um das Verhalten einer Software bei der extremen Bedingung zu analysieren. Der schwerwiegende Fehler (zum Beispiel: Programmabsturz oder Endlosschleife) kann dadurch vermieden werden.

Die Ergebnisse von Tests müssen gründlich untersucht und analysiert werden[1]

Das ist das wichtigste Prinzip. Man sollte den Fehler möglichst in der früheren Phase der Softwareentwicklung entdecken. Wenn man den Fehler in einer späteren Phase entdeckt, ist es sehr schwierig, diesen Fehler zu lokalisieren und zu reparieren. Viele Programmierer sind mit ihrem Programm so vertraut, dass sie Details in den Ergebnissen übersehen. Außerdem ist es sehr leicht, einen Fehler zu übersehen, weil die Ausdrücke von Testergebnisse sehr lang sind. Ein Fehler kommt manchmal nur in einem falschen Buchstaben vor.

Die Wahrscheinlichkeit, in einem bestimmten Segment des Programmcodes in der näheren Umgebung eines bereits bekannten Fehler weitere Fehler zu finden, ist überproportional hoch [1]

Zum Beispiel: Wenn es zwei Module A und B gibt, und der Tester hat 20 Fehler in A und 3 Fehler in B entdeckt, dann wird er mehr zusätzliche Fehler in A als in B finden. Es lohnt sich für den Tester, im Modul eines bereits bekannten Fehlers noch nach weiterem Fehler zu suchen.

Zu jedem Test gehört die Definition des erwarteten Ergebnisses vor dem Beginn des Test [1]

Wenn man das erwartete Ergebnis nicht vorher definiert, besteht die Gefahr, ein plausibles aber fehlerhaftes Ergebnis als korrekt zu betrachten. Außerdem wenn man die erwarteten Ergebnisse schriftlich vorher definiert, werden unnützlichen Diskussionen vermieden.

Ein Programm zu untersuchen, um festzustellen, ob es tut, was es tun sollte, ist nur die eine Hälfte der Schlacht. Die andere Hälfte besteht darin, zu untersuchen, ob das Programm etwas tut, was es nicht tun soll

Es ist aber auch ein Fehler, wenn das Programm das tut, was es nicht tun soll.

Testen ist definiert als die Ausführung eines Programms mit der erklärten Absicht, Fehler zu finden [1]

Es ist dem Tester gelungen, wenn er den Fehler im Programm gefunden hat. Außerdem wird seine Arbeit an der Anzahl des gefundenen Fehler gemessen. Der Tester muss einen guten Testfall entwerfen. Ein guter Testfall ist dadurch gekennzeichnet, dass er einen bisher unbekannten Fehler entdeckt.

Planen Sie nie einen Test in der Annahme, dass keine Fehler gefunden werden.

7. Zusammenfassung

Black-Box-Test und White-Box-Test werden zur Validation einer Software eingesetzt. Durch Black-Box-Test und White-Box-Test wird untersucht, ob eine Software der Spezifikation und dem Kundenwunsch entspricht oder nicht. Mit Black-Box-Test wird vor allem untersucht, ob die Software seiner Spezifikation entspricht. Es gibt drei wichtigen Methoden: Äquivalenzklassenbildung, Grenzwertanalyse und Test spezieller Werte.
Bei der Äquivalenzklassenbildung werden die Eingabedaten eines Programms in eine endliche Anzahl von Äquivalenzklassen unterteilt. Mit Hilfe der Grenzwertanalyse werden Werte an den Grenzen von Äquivalenzklassen untersucht. Äquivalenzklassenbildung und Grenzwertanalyse haben einen gemeinsamen Nachteil. Sie können keine Wechselwirkung bzw. Zusammenhänge zwischen Eingabedaten untersuchen. Mit Hilfe der Ursache-Wirkungs-Graphen wird dieser Nachteil behoben. Ein Ursache-Wirkungs-Graph ist eine formale Sprache, in die eine Spezifikation aus der natürlichen Sprache übersetzt wird. Äquivalenzklassenbildung und Grenzwertanalyse können entweder als eine Schaltung der Digitallogik oder als einen kombinatorischen logischen Graph dargestellt werden. Mit White-Box-Tests wird sichergestellt, dass das Programm keinen ungetesteten Code enthält. Bei dieser Methode wird jeder Programmpfad mindestens einmal durchlaufen.
Für die Erstellung der effektiven Testfälle zur Fehlerabdeckung ist die Kombination von Black-Box-Test und White-Box-Test zu empfehlen, da jede Methode jeweils Nachteile und Vorteile hat. Man nennt diese Kombination Broken-Box-Test oder Grey-Box-Test. Die oben beschriebenen Verfahren erhöhen die Qualität und die Sicherheit der entwickelten Software. Absolute Sicherheit kann keines der Verfahren garantieren.

Prüfe die Brücke, die dich tragen soll
Sprichwort

8. Literaturverzeichnis

[1] Georg Erwin Thaller. *Verifikation und Validation*. Vieweg, 1994.

[2] Ian Sommerville. *Software Engineering.* Addison-Wesley, 2000.

[3] Glenford J.Myers. *Methodisches Testen von Programmen.Oldenbourg,1991*.

[4] Neil Storey. *Safety-Critical Computer Systems*. Prentince Hall, 1996.

[5] Edward Kit. *Sofware Testing in The Real World*. Addison-Wesley, 1995.

[6] Ilene Burnstein. *Practical Software Testing*. Springer, 2003.

[7] Helmurt Balzert. *Lehrbuch Grundlagen der Informatik*. Spektrum, 1999.

[8] Hauptseminar Prof. Huckle : Therac 25
http://www5.in.tum.de/lehre/seminar/semsoft/unterlagen_02/therac/website

[9] Torsten Bresser. *Validieren und Verifikation (inkl. Testen, Model-Checking und Theorem Proving)*. Seminar, 2004.

[10] Friederike Nickl. *Qualitätssicherung und Testen.* sepis.

Verifikation, Validation und Testen von Sicherheitskritischen Systeme

Seminararbeit
im Fach Informatik
im Rahmen des Seminars "**Sicherheitskritische Systeme**"
an der

Universität Siegen, Fachgruppe für Praktische Informatik

eingereicht bei
Dr. Jörg Niere

vorgelegt von
Chandra Kurnia Jaya

Sommersemester 2004

Content

1: INTRODUCTION

1.1 MOTIVATION, PURPOSE OF SYSTEM VERIFICATION
1.2 SYSTEM VERIFICATION VIA MODEL CHECKING

2: THE PROCESS OF MODEL CHECKING

2.1 MODELLING
2.2 SPECIFICATION
2.3 VERIFICATION
2.4 A SIMPLE EXAMPLE WITH MICROWAVE-OVEN COOKING

3: ALGORITHMS FOR MODEL CHECKING

3.1 BASED ON AUTOMATA THEORY
3.2 BASED ON SYMBOLIC STRUCTURE
3.3 ALTERNATIVE METHODS

4: MODEL CHECKING FOR REAL-TIME SYSTEMS

4.1 DISCRETE REAL-TIME SYSTEM
4.2 CONTINUOUS REAL-TIME SYSTEM

5: MODEL CHECKING DEVELOPING TREND

6: CONCLUSION

7: REFERENCES

1: Introduction

1.1 Motivation, Purpose of System Verification

Computerised systems are more and more important to our everyday life. Digital technology is now used to supervise critical functions of Aircrafts, Trains, Nuclear Industrial Plants & Life Support Systems etc. In computer science it is important that errors must be eliminated as promptly as possible, otherwise it becomes much more expensive, because errors cost often not only the money, but also the human life… for examples:

- Pentium bug Intel Pentium chip, released in 1994 produced error in floating point division
  Cost: $475 million;
- ARIANE Failure: In December 1996, the Ariane 5 rocket exploded 40 seconds after take off
Software components threw an exception
  Cost: $400 million.
- Therac-25 Accident: A software failure caused wrong dosages of x-rays.
  Cost: Human Loss.

So the correctness of the software-based systems should be guaranteed. Currently, testing, simulation, deductive verification, and model checking are four principal techniques for ensuring the correctness of hardware and software systems.

- Simulation and Testing.

Simulation works with the abstraction of a kind of the system or a model. Compared with simulation, testing is performed on the actual product. In both cases, Simulation and Testing send the test signal at input-point and check the signal at the output-signal. However, It is not enough that we only use tests and Simulation to prove whether a system always hold itself correctly in certain situations. On the one hand they can become very expensive and on the other hand in the case of complex, asynchronous systems, these techniques can cover only a limited set of possible behaviours.

- Deductive Verification

The verification with deductive methods uses a quantity of axioms and proof rules. It is quite lengthy, very time-consuming and can only be used by experts, and therefore is rarely used.

- Model Checking

Model checking is an approach to verification where the system to be verified is viewed as a model, and the property to be proved of the system is given as a logical formula. Model Checking differs from traditional program verification methods mainly in two crucial aspects: firstly, it does not aim of being fully general and secondly it is fully algorithmic and of low computational complexity.

1.2 System Verification via Model Checking

Model checking is a general term for a collection of related formal methods, a technique offering the potential to guarantee correct functional behaviour of a system with respect to its

specification. In the last two decades model checking has emerged as a promising and powerful approach to automatic verification of systems.

Model Checking has been conceived as an automatic verification technique for finite state systems, performing an exhaustive search of the state space of the system in order to determine if some specification is true or not. Given sufficient resources, the procedure will always terminate with a yes/no answer. Since subtle errors in the design of safety critical systems that often elude traditional verification techniques can be discovered in this way, and since it has proven to be cost-efficient, Model Checking is being adopted as a standard procedure for the quality assurance of reactive systems.

Just like deductive method, the model checking is also logic-based. But contrary to deductive method, the main advantage of model checking is that, at least for finite-state systems, it can be performed fully automatically and efficiently. That means an enormous facilitation of work and saving of time. This is why now Model Checking is used by many companies such like Intel, IBM, Microsoft, Siemens.

2: The Process of Model Checking

The Process of Model Checking can be separated into modelling, specification and verification.

2.1 Modelling

First of all, we convert the system, which should be examined, into a formalism accepted by a Model Checking tool. For the modelling of systems we use finite automats. In many cases, this is only a compilation task, but we should not underestimate this task, because if we made any mistake during the conversion, it could means that the result of the verification would be worthless.

In other cases, owing to limitations on time and memory, the modelling of a design may require the use of abstraction. It is not so simply to provide this automat, because on the one hand relevant or important points must be represented in the automat, on the other hand unnecessary details should be eliminated. For example, when modelling digital circuits, it is useful to reason in terms of gates and Boolean values, rather than actual voltage levels. Likewise, when reasoning about a communication protocol we may want to focus on the exchange of messages and ignore the actual contents of the messages.

A state is a snapshot or instantaneous description of the system that captures the values of the variables at a particular instant of time. We also need to know how the state of the system changes as the result of some action of the system. We can describe the change by giving the state before the action occurs and the state after the action occurs. Such a pair of states determines a transition of the system. The computations of a reactive system can be defined in terms of its transitions. A computation is an infinite sequence of states where each state is obtained from the previous state by some transition. We use a type of state transition graph called a Kripke structure to model a system

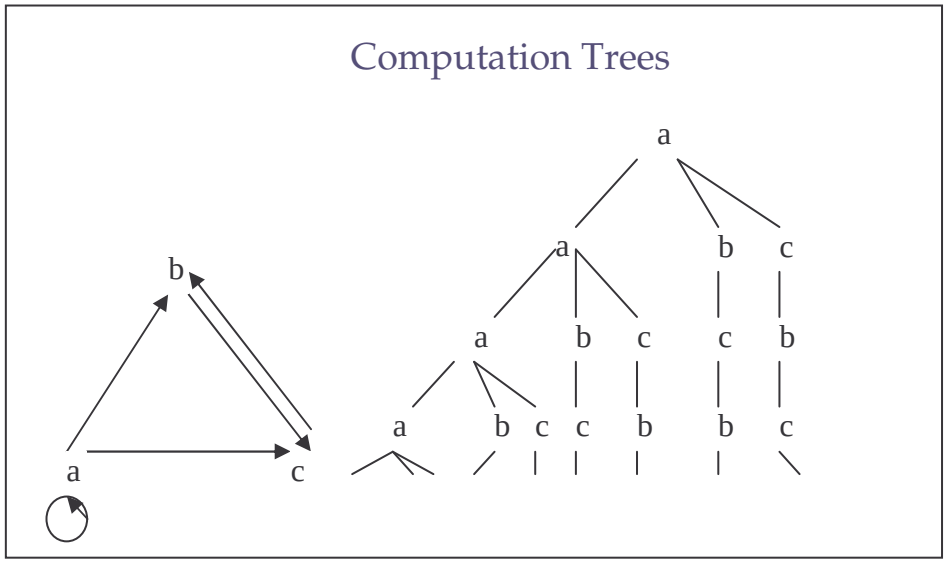
A Kripke structure is basically a graph having the reachable states of the system as nodes and state transitions of the system as edges. It also contains a labelling of the states of the system with properties that hold in each state.

Definition: Let AP be a non-empty set of atomic propositions. A Kripke structure over a set of atomic propositions AP is a four-tuple
M = (S, S0, R, L) where

• S is a finite set of states.
• S0 ⊆ S is the set of initial states.
• R ⊆ S × S is a transition relation
• L: S→2(AP) is a function that labels each state with the set of atomic propositions true in this state.

Computation trees:

• State Transiton Graph or Kripke Model
• Unwind State Graph to obtain Infinite Tree



2.2 Specification

The second step is specification, that means: we give the properties that the design must satisfy. Usually this is achieved by using Temporal Logic-- A logic with a notion of time included. (In 1980s, a verification technique called Temporal Logic Model Checking was developed independently by Clarke and Emmerson in the USA and Queille and Sifakis in France.)

• Classical Logic: with its basic boolean operators ∨(disjunction), ∧(conjunction), ¬(negation), →(implication) and ↔(equivalence). We also have the first order operators ∃(existence), and ∀(universality). Good for describing static conditions

• Temporal Logic: an extension of Classical logic, which incorporates special operators, that cater for time. With Temporal Logic we can specify how components, protocols, objects, modules, procedures and functions behave as time progresses. The specification is done with Temporal Logic statements that make assertions about properties and relationships in the past, present, and the future. It is convenient to use Temporal Logic to express temporal

relationships between events and so Temporal Logic is widely used for specification formulas. The formulas are interpreted over Kripke structures, which can model computation; Therefore, Temporal Logic is very useful in formal verification.

There are five basic temporal operators for the Temporal Logic to describes how static conditions change over time.

• X (‘‘next time’’) requires that a property holds in the next state of the path.
• F (‘‘in the future’’) requires that a property will hold at some state on the path.
• G (‘‘globally’’) requires that a property holds at every state on the path.
• U (‘‘until’’) combines two properties. It holds if there is a state on the path where the second property holds, and at every proceeding state on the path, the first property holds.
• R (‘‘Release’’) is the logical dual of the U operator. It requires that the second property holds along the path where the second property holds. However, the first property is not required to hold eventually.

But these 5 operators can be used to describe properties of only one path through the tree, so there are two quantifiers for the Temporal Logic to describe the branching structure in the computation tree. With these two path quantifiers it is possible that describe properties of all of the paths or some of the paths through the tree.

• A (‘‘always’’) for all computation paths
• E (‘‘exists’’) for some computation paths

Three are three main ways to represent Temporal Logic:

• CTL* (Computation Tree Logic*): a powerful and complex logic. Often we need only the subset (fragment) of CTL*, such as LTL or CTL, in order determined properties specify.

• CTL (Computation Tree Logic) Branching time: describes all possible time lines. CTL has the same operators like CTL*, however it has syntactic restrictions, it means that each temporal logical operator must directly follow one of these two path quantifiers (A or E). So for the CTL we have the following 10 basis operators: AX and EX; AF and EF; AG and EG; AU and EU; AR and ER.

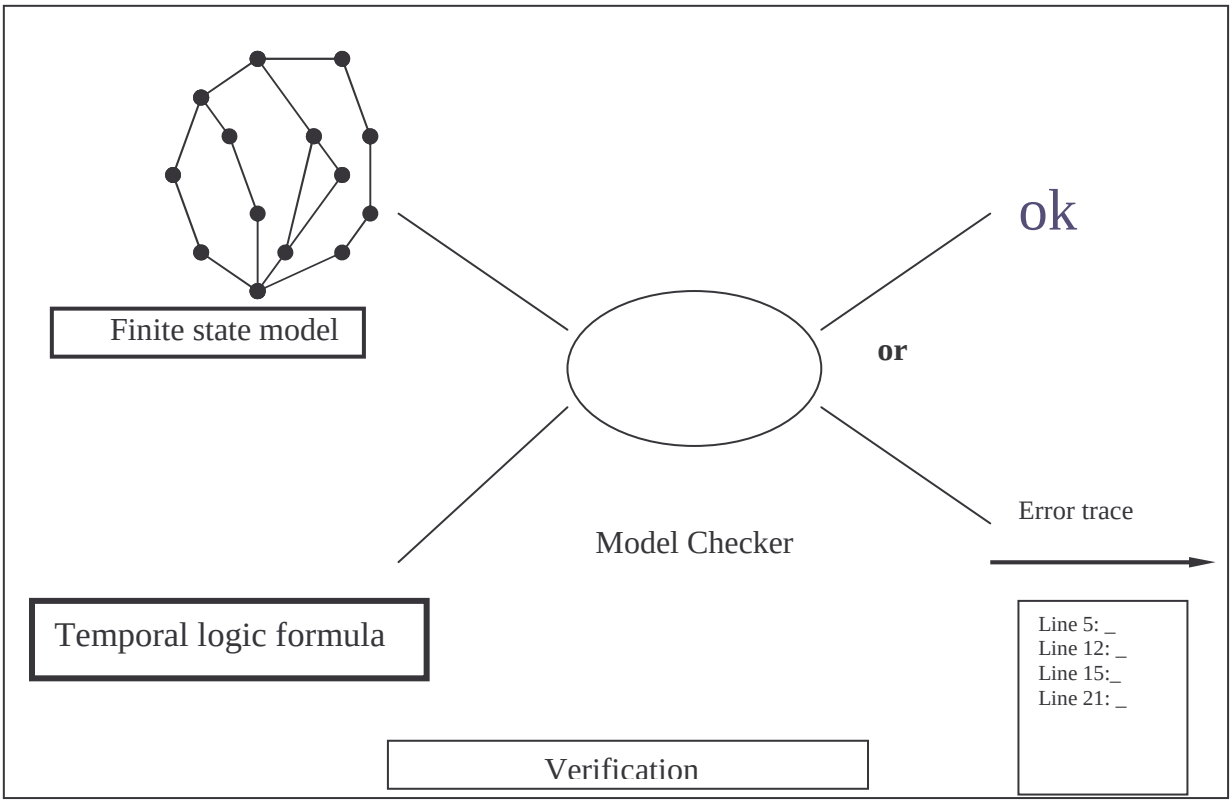
• LTL (Linear Temporal Logic) Linear time: describes single possible time line, All LTL formulas are properties of paths (as opposed to states). An LTL formula is interpreted with respect to a fixed path. Compare with CTL, the LTL is perhaps more intuitive and can express some properties CTL cannot (and vice versa). We can translate LTL to Buchi-automata. And it is Linear because conceptually it’s only about paths, as opposed to computation trees (CTL, where paths can branch).

These 3 different variants of the Temporal Logic differ particularly in the availability of the temporal operators and path quantifiers as well as in the semantics of these operators. So they will be applied in different ranges of hardware or software application-area.

An important issue in specification is completeness. Model checking provides means for checking that a model of the design satisfies a given specification, but it is impossible to determine whether the given specification covers all the properties that the system should satisfy.

2.3 Verification

After we finish all preparations, the Model Checking can begin. Now one must decide whether for the given model of a system, the Kripke structure and the specification (with Temporal Logic) are valid. According to temporal logic, we have CTL* -Model-Checking, CTL-Model-Checking and LTL-Model-Checking.



Ideally the verification is completely automatic. However, in practice it often involves human assistance. One such manual activity is the analysis of the verification results. In case of a negative result, the user is often provided with an error trace. This can be used as a counterexample for the checked property and can help the designer in tracking down where the error occurred. In this case, analysing the error trace may require a modification to the system and reapplication of the Model Checking algorithm.

An error trace can also result from incorrect modelling of the system or from an incorrect specification (often called a false negative). The error trace can also be useful in identifying and fixing these two problems. A final possibility is that the verification task will fail to terminate normally, due to the size of the model, which is too large to fit into the computer memory. In this case, it may be necessary to redo the verification after changing some of the parameters of the model checker or by adjusting the model.

2.4 A simple example with microwave-oven cooking

Here is a simple example with microwave-oven cooking for the Process of Model Checking:

Step 1:

Modelling with the Kripke-Structure M = (S, S0, R, L)

• S = (S1, S2, S3, S4)
• S1 is the initial state
• R = ({S1, S2} {S2, S1}, {S1, S4}, {S4, S2}, {S2, S3}, {S3, S2}, {S3, S3}
• L (S1) = {¬close, ¬ start, ¬ cooking} L (S2) = {close, ¬ start, ¬ cooking} L (S3) = {close, start, cooking} L (S4) = {¬close, start, ¬ cooking}

The Kripke-Structure M describes the states (S1, S2, S3, S4) of a microwave-oven.

The initial state S1 – means the door is open; the microwave-oven does not work. After someone closes the door, state S1 will go to state S2, at that time if someone pushes the start-up button, the microwave-oven will begin to work and go to state S3. It is not allowed that the microwave-oven starts to work before he closes the door, because it is very dangerous. But if someone pushes the start-up button before he closes the door, the microwave-oven will not work and go to state S4. Only after he closes the door, the microwave-oven will go to state S1 and in state S1 the microwave-oven will begin to work only after he pushes the start-up button again. After cooking, the microwave-oven will go from state S3 back to state S2. And after opening the door, the microwave-oven will go from state S2 back to state S1.

Figure: Automat --- Graph of the Kripke-Structure M of microwave-oven:

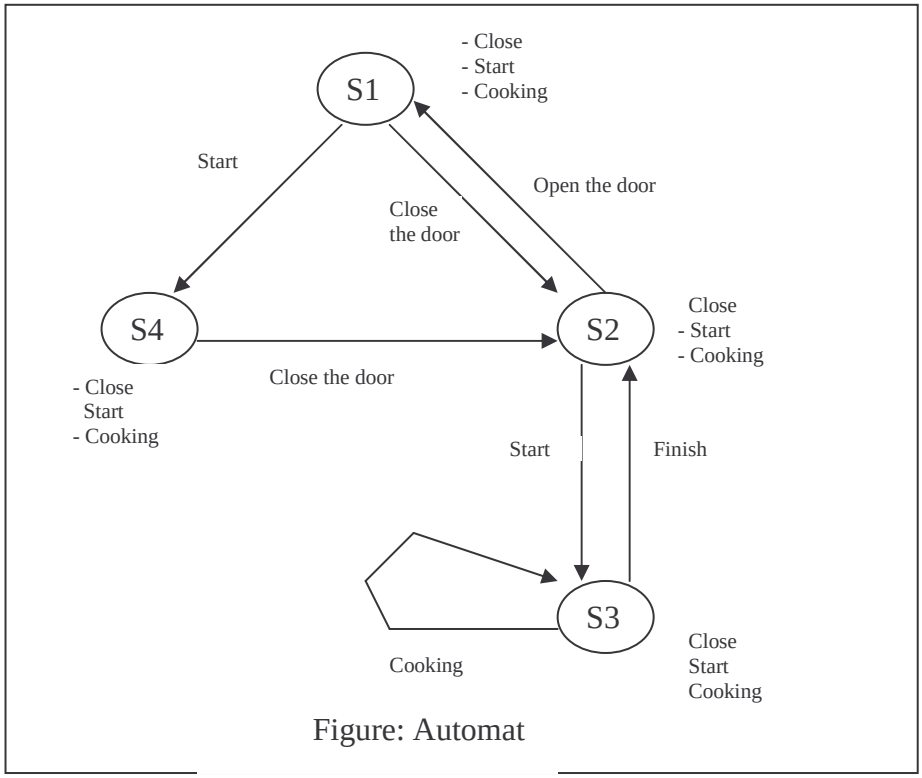


Figure: Automat

Step 2:

Specification with CTL-Formal -- possible requirements of the microwave oven:

1. Every time after someone pushes the start button, the microwave oven begins to cook.
↔ AG (start ⇒ AF cooking)

2. If the door is closed and someone pushes the start button, the microwave oven begins to cook.
$\leftrightarrow$ AG ((close $\wedge$ start) $\Rightarrow$ AF cooking)

Step 3:

To the first CTL-Formal: Every time after someone pushes the start button, the microwave oven begins to cook. $\leftrightarrow$ AG (start $\Rightarrow$ AF cooking)

Now we describe the CTL formulas in another way that only path quantifiers and the classical logic operators in the formulas. This step serves for the better and more efficient execution of the algorithm.

Then, for the first CTL-Formal we get: AG (start $\Rightarrow$ AF cooking) $\equiv$ AG ($\neg$ start $\vee$ AF cooking)

Through 2 times negating we get: $\neg\neg$AG ($\neg$ start $\vee$ AF cooking)

Because of $\neg AG\phi \equiv EF \neg\phi$, we get: $\equiv \neg$EF (start $\wedge$ EG $\neg$ cooking))

Now we have the correct form. We divide the CTL formula into several partial formulas and the conditions in those partial formulas are true. We begin from simple partial formulas to the more complicated formulas, until all of the formulas are true.

- S (start) = {S3, S4}
- S ($\neg$cooking) = {S1, S2, S4}
- S (EG $\neg$ cooking) = {S1, S2, S4} (all conditions lie on a path)
- S (start $\wedge$ EG $\neg$ cooking) = {S4}
- S (EF (start $\wedge$ EG $\neg$ cooking)) = {S1, S2, S3, S4} (can be followed with S4)
- S ($\neg$ (EF (start $\wedge$ EG $\neg$ cooking))) = {}

The starting condition S1 does not lie in this formula, so the formula is not valid. And in the fact the microwave-oven works like that, because it is not allowed that the microwave-oven starts to work before someone closes the door for safety reasons. --- Therefore, the model does not fulfill this specification!

The model should however absolutely fulfill the second CTL formula, otherwise we could not use the microwave-oven.

To the second CTL-Formal: If the door is closed and someone pushes the start button, the microwave oven begins to cook. $\leftrightarrow$ AG ((close $\wedge$ start) $\Rightarrow$ AF cooking)

AG ((close $\wedge$ start) $\Rightarrow$ AF cooking) $\equiv$AG($\neg$ (close $\wedge$ start) $\vee$ AF cooking)$\equiv$AG($\neg$ close $\vee$ $\neg$ start $\vee$ AF cooking)$\equiv$ $\neg\neg$AG($\neg$ close $\vee$ $\neg$ start $\vee$ AF cooking)$\equiv\neg$ EF(close $\wedge$ start $\wedge$ EG $\neg$ cooking)

Now the algorithm can be applied to the formula and we get:

- S (close)= {S2, S3}
- S (start)= {S3, S4}
- S ($\neg$ cooking) = {S1, S2, S4}
- S (EG $\neg$ cooking) = {S1, S2, S4}
- S (close $\wedge$ start $\wedge$ EG $\neg$ cooking) = {}
- S (EF (close $\wedge$ start $\wedge$ EG $\neg$ cooking) = {}
- S ($\neg$ (EF (close $\wedge$ start v EG $\neg$ cooking)) = {S1, S2, S3, S4}

Since the initial condition S1 is contained in this quantity, it means that our microwave-oven, described by the Kripke structure, fulfills specification…Therefore, it is correct!

3: Algorithms for Model Checking

The initial implementation of the Model Checking problem set an explicit representation of the Kripke structure as a labelled, directed graph. For systems with limited number of processes, where the number of possible states was rather small, this approach proved to be practical. However, this method was unable to handle extensive number of states generated by complex concurrent systems. One of the major problems in the application of model checking techniques to practical verification problems is so-called state explosion problem. The state-space explosion problem is a problem that not only appears for model checking, but also for various other problems like performance analysis that are strongly based on state space exploration.

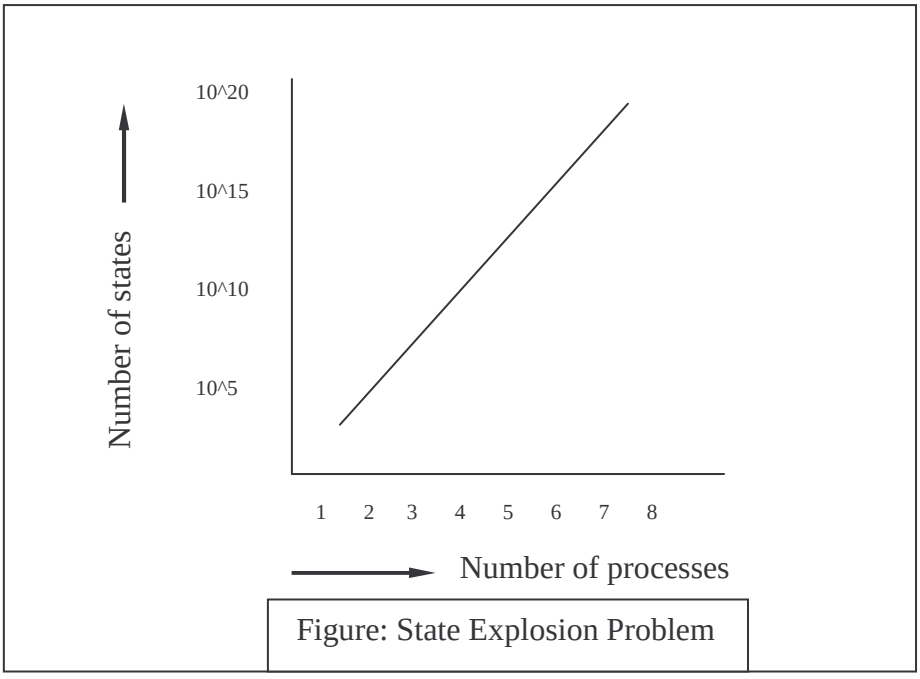


Figure: State Explosion Problem

Model Checking is increasingly used in the formal verification of hardware and software. For concurrent system with small numbers of processes, the number of states was usually fairly small, and the approach was often quite practical. In systems with many concurrent parts however, the number of states in the global state transition graph was too large to handle. (See the figure: Number of states typically grows exponentially in the number of process) Much work over the past 10-15 years has gone into overcoming state-space explosion. Let us see some of the major techniques for tackling this problem.

3.1 Based on Automata Theory

3.1.1 On the Fly Technology

Explicit state model checking uses a graph to represent a Kripke structure with nodes for states and edges for transitions. Extracting this structure from a concurrent system, prior to

model checking, may result in a graph that is exponentially bigger than the system. Algorithms based on the explicit state enumeration could be improved if only a fraction of the reachable pairs are explored

A finite automaton is a mathematical model of a device that has a constant amount of memory, independent of the size of its input. Formally, a finite automaton A is a five tuple ($\sum$, Q, Δ, Q0, F) with :

- $\sum$ is the finite alphabet.
- Q is the finite set of states.
- $\Delta \subseteq Q \times \sum \times Q$ is the transition relation.
- $Q0 \subseteq Q$ is the set of initial states.
- $F \subseteq Q$ is the set of final states.

By using the automata theoretic approach to model checking, it is possible in many cases to avoid construction the entire state space of the modelled system. This is because the states of the automaton are generated only when needed, while checking the emptiness of its intersection with the property automaton S. This tactic is called “on-the-fly” mode checking. The modelled system is converted into a corresponding Büchi automaton A, and the negation of the specification φ is translated into another automaton S. Then, the emptiness of the intersection of A and S is checked. If the intersection is not empty, a counterexample is reported. Instead of constructing the automata for both A and S first, we will only construct the property automaton S. We then use it to guide the construction of the system automaton A while computing the intersection. In this way, we may frequently construct only a small portion of the state space before we find a counterexample to the property being checked.

One advantage of on-the-fly model checking is that when computing the intersection of the system automaton A with the property automaton S, some states of A may never be generated at all. Another advantage of the on-the-fly procedure is that a counterexample may be found before completing the construction of the intersection of the two automata. Once a counterexample has been found and reported, there is no need to complete the construction.

3.1.2 Partial-Oder Reduction Technology

Systems that consist of a set of components that cooperatively solve a certain task are quite common in practice, e.g. hardware systems, communications protocols, distributed systems, and so on. Typically, such systems are specified as the parallel composition of n processes. The state space of this specification equals in worst case the product of the state spaces of the components. A major cause of this state space explosion is the representation of parallel is by means of interleaving. Interleaving is based on the principle that a system run is a totally ordered sequence of actions. In order to represent all possible runs of the systems, all possible interleaving of actions of components need to be represented. For checking a large class of properties, however, it is sufficient to check only some representative of all these interleaving.

For example, if two processes both increment an integer variable in successive steps, the end result is the same regardless of the order in which these assignments occur. The underlying idea of this approach is to reduce the interleaving representation into a partial-order representation. System runs are now no longer totally ordered sequences, but partially ordered

sequences. The partial order reduction is aimed at reducing the size of the sate space that needs to be searched by model checking algorithms. It exploits the commutativity of concurrently executed transitions, which result in the same state when executed in different orders.

The method consists of constructing a reduced state graph. The full state graph, which may be too big to fit in memory, is never constructed. The behaviours of the reduced graph are a subset of the behaviours of the full state graph. The justification of the reduction method shows that the behaviours that are not present do not add any information. More precisely, it is possible to define an equivalence relation among behaviours such that the checked property cannot distinguish between equivalent behaviours. If a behaviour is not present in the reduced state graph, then an equivalent behaviour must be included.

There are basically three different techniques for partial-order approaches towards model checking.

• In dynamic partial- order reduction a subset of the states that need to be explored is determined during the search procedure. This embodies that the possible interesting successors of a given state are computed while exploring the current state. Dynamic partial-order reduction is the most popular and most developed technique; it requires, however, a modification of the search procedure.

• Static partial-order reduction techniques avoid the modification of the search engine of the model checker by performing the reduction at the level of the system specification, that is, syntactically. The subset of the states that need to be explored is thus determined before the actual searching takes place.

• In the purely partial-order approach, the construction of the interleaved expansion of parallel components is avoided by directly constructing a partial-order representation of the state space from the system specification. An interleaving representation is thus never generated, and needs never to be stored.

3.2 Based on Symbolic Structure

In the fall of 1987, McMillan, a graduate student at Carnegie Mellon University, realized that by using a symbolic representation for the state transition graphs, which based on the use of BDD (Binary Decision Diagrams), much larger systems could be verified.

A state of the system is symbolically represented by an assignment of boolean values to the set of state variables. A Boolean formula (and thus its BDD) is a compact representation of the set of the states represented by the assignments, which make the formula true. The transition relation can therefore be expressed as a boolean formula in terms of two sets of variables, one set encoding the old state and the other encoding the new. This formula is then represented by a binary decision diagram. Then, Model Checking algorithm is based on computing fixpoints (The fixpoints are sets of states that represent various temporal properties of the concurrent system.) of predicate transformers that are obtained from the transition relation. In the new implementations, both the predicate transformers and the fixpoints are represented with BDD. Therefore, the explicit construction of the global state graph for the entire concurrent system could now be avoided.

The BDD technique is a technique to encode transition systems in terms of a compact model that is suite to represent boolean functions efficiently. In a binary decision diagram the redundancy that is present in a decision tree is reduced. A binary decision diagram is a binary decision tree in which isomorphic subtrees are collapsed and redundant vertices are omitted. Because the symbolic representation captures some of the regularity in the state space determined by circuits and protocols, it is possible to verify systems with an extremely large number of states. The BDD technique cannot guarantee the avoidance of the state space explosion in all cases, but provide a compact representation of several systems, allowing these systems to be verified.

Later on, McMillan developed a model checking system called SMV, which is based on a language for describing hierarchical finite-state concurrent systems. Programs in the language can be annotated by specifications expressed in temporal logic. The model checker extracts a transition system represented as an OBDD (Bryant's ordered binary decision diagrams) from a program in the SMV language and uses an OBDD-based search algorithm to determine whether the system satisfies its specification. The same as most of the model checking systems, if it determines that a certain property has not been satisfied, it will produce the counterexample proving the invalidity of the specification.

Symbolic Model Checking derives its power from efficient data structures for representation and manipulation of large sets of states. By using the original CTL model checking algorithm of Clarke and Emerson with the new representation for state transition graphs, it became possible to verify some examples that had more than 10^20 states. Since then, various refinements of the OBDD-based techniques by other researchers have pushed the state count up to more than 10^120.

The SMV system has been widely distributed, and a large number of examples have now been verified with it. Some examples provide convincing evidence that SMV can be used to debug real industrial designs : In 1992 Clarke and his students used SMV to verify the IEEE Future+ cache coherence protocol. They found a number of previously undetected errors in the design; In 1992 Dill and his students found several errors, ranging from initialized variables to subtle logical errors during the verification of the cache coherence protocol of the IEEE Scalable Coherent Interface.

3.3 Alternative Methods

Symbolic representations and partial-order reduction have greatly increased the size of the systems that can be verified to 10^20 – 10^30 states, but still, there are a vast number of systems that are too complex to handle (Typical verification tasks still take modern sequential computers to their memory limits or use the accumulated memory of parallel computers)

It is important to find techniques that can be used in conjunction with existing approaches have been introduced, allowing for complex systems to be model checked. Five such techniques are equivalence, compositional reasoning, abstraction, symmetry and induction.

3.3.1 Equivalence

Equivalence is a technique to transform models into equivalent, but smaller models that satisfy the same properties, that means replace a large structure by a smaller structure, which satisfies the same properties to avoid the state explosion problem. Most precisely, Given a logic L and a structure M, try to find a smaller structure M' that satisfies exactly notion of equivalence between structures that can be efficiently computed and guarantees that two structures satisfy the same set of formulas in L.

Simulation equivalence and bisimulation equivalence are some of most common notions of semantic equivalence. Informally speaking two models are bisimilar if each transition in the once model can be simulated by the other and vice versa.

3.3.2 Compositional Reasoning

Compositional verification is a technique to decompose properties in sub-properties such that each sub-property can be checked for a part of the state space, and the combination of the sub-properties implies the required global property. Efficient algorithms for compositional verification can extend the applicability of formal verification methods to much larger and more interesting systems. The compositional Reasoning technique exploits the modular structure of systems, composed of multiple processes running in parallel. To simplify the specification of such a system, the specifications for such systems can often be decomposed into properties that describe the behaviour of small parts of the system. An obvious strategy is to check each of the local properties using only the part of the system that it describes. Then, the complete system must satisfy the overall specification: if each local property holds and if the conjunction of the local properties holds.

For Example, we verify a communication protocol that is modelled by three definite state processes: a transmitter, some type of network, and a receiver. Suppose that the specification for the system is that data is eventually transmitted correctly from the sender to the receiver. Such a specification might be decomposed into three local properties. First the data should eventually be transmitter to the network. Second, the data should eventually be transferred correctly from one end of the network to the other. Third, the data should eventually be transferred correctly from the network to the receiver. We might be able to verify the first of these local properties using only the transmitter and the network, the second using only the network and the third using only the network and the receiver. By decomposing the verification in this way, we never have to compose all of the processes and therefore avoid the state explosion phenomenon.

When this form of compositional reasoning is not feasible because of mutual dependencies between the components, a more complex strategy is necessary. In such cases, when verifying a property of one component we must make assumptions about the behaviours of the other components. The assumptions must later be discharged when the correctness of the other components is established. This strategy is called assume-guarantee reasoning.

3.3.3 Abstraction

Abstract interpretation is a technique to reduce the increase of the state space due to date by providing a mapping between the concrete date values in the system and a small set of abstract data values that suffices to check the properties of interest. The abstract system is often much smaller than the actual system, and as a result, it is usually much simpler to verify properties at the abstract level.

For example, in verifying the addition operation of a microprocessor, we might require that the value in one register is eventually equal to the sum of the values in two other registers. In such situations abstraction can be used to reduce the complexity of model checking.

There are two different abstraction techniques: The cone of influence reduction and data abstraction. Both of these techniques are performed on a high level description of the system, before the model for the system is constructed. Therefore, we avoid the construction of the unreduced model that might be too big to fit into memory.

3.3.4 Symmetry

Symmetry reduction approach applies to systems that are composed of components that are replicated in a regular manner. It is possible to find symmetry in memories, caches, register files, bus protocols, network protocols—anything that has a lot of replicated structure. These reduction techniques are based on the observation that having symmetry in the system implies the existence of nontrivial permutation groups that preserve both the state labelling and the transition relation. Such groups can be used to define an equivalence relation on the state space of the system. The model induced by this relation is often smaller than the original model.

For example, a large number of protocols involve a network of identical processes communicating in some fashion. Hardware devices contain parts such as memories and register files that have many replicated elements. These facts can be used to obtain reduced models for the system. Having symmetry in a system implies the existence of a nontrivial permutation group that preserves the state transition graph. Such a group can be used to define an equivalence relation on the state space of the system and to reduce the state space. The reduced model can be used to simplify the verification of properties of the original model expressed by a temporal logic formula.

3.3.5 Induction

Induction involves reasoning automatically about entire families of finite-state systems. Such families arise frequently in the design of reactive systems in both hardware and software.

Typically, circuit and protocol designs are parameterized, that is, they define an infinite family of systems. For example, a circuit designed to add two integers has the width of the integers n as a parameter; a bus protocol may be designed to accommodate an arbitrary number of processors, and a mutual exclusion protocol can be given for a parameterized number of processes. We would like to be enable to check that every system in a given family satisfied some temporal logic property. In many interesting cases, it is possible to provide a form of invariant process that represents the behavior of an arbitrary members of the family. Using this invariant, we can then check the property for all of the members of the family at once. An inductive argument is used to verify that the invariant is an appropriate representative.

4: Model Checking for real-time Systems

Computers are frequently used in critical applications where predictable response times are essential for correctness. Such systems are called real-time systems. Examples of such applications include controllers for aircraft, industrial machinery and robots. Due to the nature of such applications, errors in real-time systems can be extremely dangerous, even fatal. Guaranteeing the correctness of a complex real-time system is an important and nontrivial task. Because of this, only conservative and usually special approaches to design and implementation are routinely used.

Some factors make the validation of real-time systems particularly difficult. The architecture of computer applications is becoming extremely complicated. As a system increases in complexity, so does the probability of introducing an error. Moreover, performance is becoming an important factor in the success of new applications. Due to competition, new products have to fully utilize the available resources. A slow component can affect the performance of the whole system. Consequently, the task of verifying that new applications satisfy their timing specifications is more critical than ever before.

4.1 Discrete real-time System

When time is discrete, possible clock values can only occur at integer time values. This type of model is appropriate for synchronous systems, where all of the components are synchronized by a single global clock. The duration between successive clock ticks is chosen as the basic unit for measuring time. This model has been successfully used for reasoning about the correctness of synchronous hardware designs for many years. Real-time concurrent systems are particularly difficult to verify because their correctness depends on the actual times at which events occur.

An example of such a technique is static time slicing, which divides time equally among all tasks. Each task executes until its time slot has been used and then release the processor. The resulting problem is easy to analyse, but rather inefficient, since all tasks are given equal resources, regardless of their importance or resource utilization.

Rate-monotonic scheduling theory (RMS) is a more powerful technique to analyse the behaviour of a real-time system. The RMS theory is applicable to systems described by a set of periodic tasks. Each task corresponds to a concurrent process of the system and is characterized by its periodicity (how often it executes) and its executive time at each instantiation.

4.2 Continuous real-time System

Continuous time is the natural model for asynchronous systems, because the separation of events can be arbitrarily small. This ability is desirable for representing causally independent events in an asynchronous system. Moreover, no assumptions need to be made about the speed of the environment when this model of time is assumed.

In order to model asynchronous systems using discrete time, it is necessary to separate time by choosing some fixed time quantum so that the delay between any two events will be a multiple of this time quantum.

5: Model Checking developing Trend

There are a number of ways that model checkers can be improved to make them easier to use by engineers.

1) Relatively straightforward extensions of current systems: one problem with current systems is how to make the specification language more expressive and easier to use. Some type of timing diagram notation may be more natural for engineers than CTL.

2) Require more theoretical work: one direction for research is to develop even more concise techniques for representing Boolean functions. When better representations are developed, they can easily be incorporated into model-checking algorithms.

3) Use abstraction and compositional reasoning techniques to handle more complex systems: There have been some attempts to automate abstraction techniques for hardware. Some work has also recently appeared on automatic application of compositional methods.

4) probabilistic verification: It may be possible to tell not only whether a failure is possible in a system, but also what the probability of the failure occurring is. If such an analysis were performed, failures with extremely low probabilities of actually occurring could be safely ignored.

5) The ability to reason automatically about entire families of finite-state systems: Such families arise frequently in the design of reactive systems in both hardware and software. We have focused on finite model structures, but recent research shows that effective Model Checking is possible also for certain classes of finitely presented infinite structures.

6) Investigation Model Checking techniques combined with theorem proving: Both model checkers and theorem proves will be needed to establish the correctness of complex circuits. Theorem provers seen necessary for reasoning about those parts of a complex microprocessor like the floating-point arithmetic unit that require relatively deep mathematical knowledge. The problem is how to combine the two very different styles of reasoning into a single framework so that a user can smoothly integrate the results obtained by each.

6: Conclusion

In this article we give a brief overview of the main system verification techniques, with a particular focus on the model-checking technique. Model checking is an important technique to verify the correctness of software and hardware systems. Validation of model checkers is a very important issue due to their automatic nature. It can ensure the correctness of critical application such as communication protocol or digital circuit. In this article, two main tactics separately based on automata theory and symbolic structure for concurrent systems are described and the methods to solve state explosion problem are given. Then the methods of model checking for real-time systems is introduced and developing trend of Model Checking is analysed.

The techniques described in this article have already been used to find nontrivial errors in circuit and protocol designs. Current model checking tools work sufficiently well to be of use in industry, and indeed, a number of companies such as AT&T, Fujitsu, Intel, IBM, Lucent, Motorola, and Siemens are beginning to use model checking as part of their design process.

7: References

1. Dong Wei, Wang Li, and Qi Zhi-Chang, Journal of computer research & development, June 2001

2. Dr. H, Giese, URL: http://www.uni-paderborn.de, Vorlesung Safety-Critical Computer Systems, Techniques for Safety, Software Model Checking, 2003

3. Markus Müller-Olm, David Schmidt, and Bernhard Steffen, Model-Checking: A Tutorial Introduction, 1999

4. Javier Esparza, David Hansel, Peter Rossmanith, and Stefan Schwoon, Efficient Algorithms for Model Checking Pushdown Systems, URL: www.sinc.sunysb.edu/Stu/pyang/slides/pushdown.ppt, June, 2001

5. Dragana Cvijanovic, Model Checking, URL: www.cs.ucl.ac.uk/staff/W.Emmerich/ lectures/3C05-02-03/aswe17-essay.pdf, 2001

6. S. Merz, Model Checking: A tutorial Overview. Institut fur Infomatik, Universitat Munchen, 2001

7. Edmund M. Clark, Jr., Orna Grumberg, and Doron A. Peled, Model Checking, 1999

8. Neil Immerman, Model Checking, URL: http://www.cs.umass.edu/~immerman/modelCheck/, 2004

9. Martin Leucker and Rafa Somla, Parallel Model Checking for LTL, CTL* , and L 2, Electronic Notes in Theoretical Computer Science 89 No. 1 (2003) URL: http://www.elsevier.nl/locate/entcs/volume89.html, 2003

10. Joost-Pieter katoen, Concepts, Algorithms, and & Tool for Model Checking, Friedrich-Alexander Universitat Erlangen – Nuernberg, 1999

11. Prof. Dr. Wolfgang Thomas, RWTH Aachen, Vorlesungsmaterial - "Model-Checking", WS 2003/04, URL: http://www-i7.informatik.rwth-aachen.de/d/teaching/ws0304/modelchk/

12. Prof. Dr. Javier Esparza, Vorlesung Model Checking (SS 2003) URL: http://www.informatik.uni-stuttgart.de/fmi/szs/teaching/ss2003/modelchecking/folien.pdf

13. Model Checking (Überblick), URL: http://www.software-kompetenz.de/?2885

14, Prof. Dr. H. Peter Gumm, Vorlesungsmaterial: Model Checking , URL: http://www.mathematik.uni-marburg.de/~tschroed/vor01SSModelChecking.html

15. Definition of model checking from NIST, URL: http://www.nist.gov/dads/HTML/modelcheckng.html

16. Fakultät für Informatik der Technischen Universität München,Hauptseminar im SS 2002 Model Checking, URL: http://wwwbrauer.informatik.tu-muenchen.de/seminare/model-checking/SS2002/

End Of Document

UML 2.0 und die Modellierung von Realtime-Systemen

Seminararbeit
im Fach Informatik
im Rahmen des Seminars "**Sicherheitskritische Systeme**"
an der

Universität Siegen, Fachgruppe für Praktische Informatik

eingereicht bei
Dr. Jörg Niere

vorgelegt von
Christoph Hardt

Sommersemester 2004

Inhaltsverzeichnis

1. Einleitung
2. Neuerungen der UML 2.0
3. Interaktionsdiagramme
4. Modellierung von Interaktionen
5. Timing-Diagramme
    - 5.1 Überblick
    - 5.2 Anwendungsbeispiel
    - 5.3 Anwendung im Projekt
    - 5.4 Notationselemente
        - 5.4.1 Interaktion / Interaktionsrahmen
        - 5.4.2 Lebenslinie
        - 5.4.3 Zeitverlaufslinie
        - 5.4.4 Nachricht
        - 5.4.5 Sprungmarke
        - 5.4.6 Wertverlaufslinie
6. Fazit
7. Literaturverzeichnis
8. Abbildungen

1. Einleitung

Die UML 2.0 ist das Ergebnis eines mehrjährigen Entwicklungsprozesses der Mitte 2001 begann und vermutlich bis Ende 2004 mit der Verabschiedung der endgültigen Spezifikationen abgeschlossen sein wird. Mit der UML 2 sollen einige grundlegende Änderungen verwirklicht werden. Ein Ziel der Überarbeitung war eine Reduzierung der Komplexität und des Umfangs um die UML2 schneller zu erlernen und angemessen zu benutzen. Außerdem sollte auf die Veränderungen im Softwareentwicklungsmarkt der letzen Jahre eingegangen werden. Zusätzlich sollten neue Anwendungsbereiche erschlossen werden, wie zum Beispiel die Modellierung von technischen Systemen mit Echtzeitanforderungen.
Aus den Anforderungen entstanden hauptsächlich zwei Dokumente – Infrastructure [6] und Superstructure [7] – die sich gegenseitig ergänzen. Grundlegende Sprachkonstrukte und die Basisarchitektur wurden in der Infrastructure [6] festgehalten. Die darauf aufbauenden Diagrammnotationen und die Semantik stehen in der Superstructure [7].

2. Neuerungen in der UML 2.0

Die UML 2.0 besitzt 13 Diagrammtypen, die größtenteils schon in den 1.x Versionen vorhanden waren und unterschiedlich starken Veränderungen unterzogen wurden. Komplett neu sind drei Diagramme: das Timing-Diagramm, das später noch ausführlich behandelt wird, das Interaktionsübersichtsdiagramm und das Kompositionsstrukturdiagramm. Abbildung 1 zeigt eine Übersicht über alle Diagrammtypen der UML

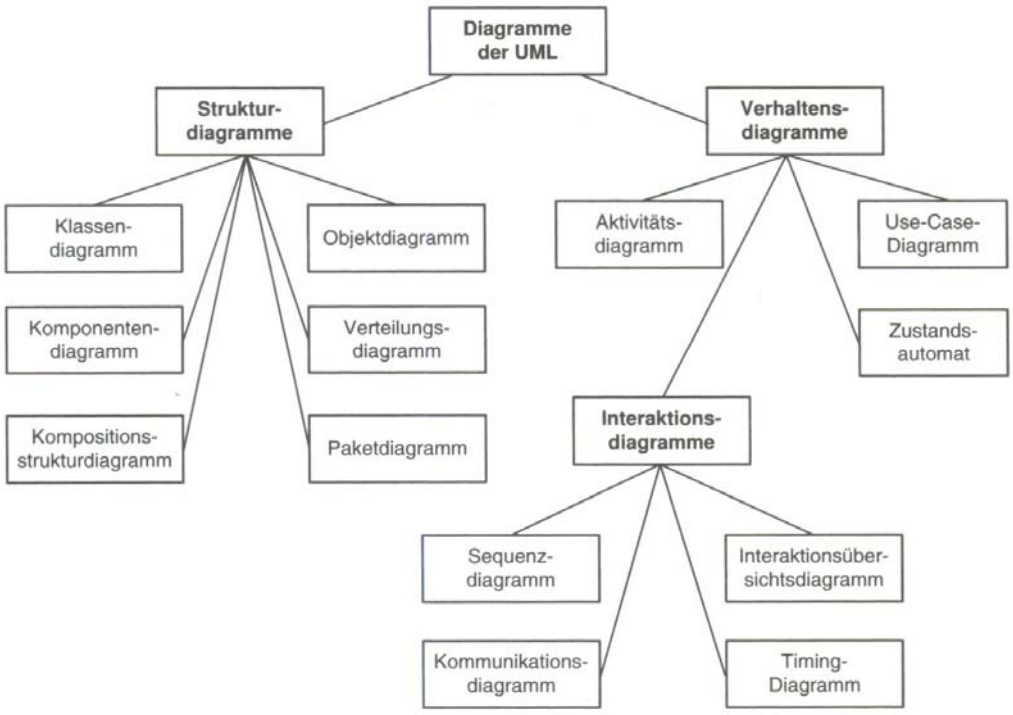


Abbildung 1: Die Diagrammtypen der UML 2 [1]

Neuerungen in der UML 2 im Überblick:

- Die meisten Änderungen haben die Aktivitätsdiagramme erfahren. Sie basieren nun auf erweiterten Petri-Netzen. Durch die Änderungen erhöht sich die Flexibilität bei der Flussmodellierung.

- Für Verhaltensdiagramme steht alternativ zur grafischen Darstellung nun eine tabellarische Darstellung zur Verfügung.

- Das Kollaborationsdiagramm wurde in Kommunikationsdiagramm umbenannt.

- Bei Klassendiagrammen wurden einige „Unsauberkeiten“ entfernt. Sie bleiben allerdings größtenteils unverändert.

- Sequenzdiagramme sind strukturier- und zerlegbar

- Verteilungsdiagramme zeigen die Laufzeitaspekte einer Architekturumsetzung. (Kommunikationsbeziehungen zwischen Einzelkomponenten)

- Das neu eingeführte Kompositionsstrukturdiagramm zeigt Interna von Systemteilen und deren Zusammenspiel in Form von Laufzeitinstanzen.

- Ebenfalls neu eingeführt wurden die aus der Elektrotechnik bekannten Timing-Diagramme um eine präzise Beschreibung des Zeitverhaltens von Objekten und Systemen zu ermöglichen.

- Das dritte neu eingeführte Diagramm, das Interaktionsübersichtsdiagramm, dient zur Darstellung der Abfolge mehrerer Verhaltensmodelle.

- Durch die Änderungen im Komponentendiagramm wird die Modellierung gängiger Marktstandards (.NET und J2EE) besser unterstützt.

- Die Zustandsautomaten ermöglichen eine verbesserte Verknüpfung von statischen Elementen und dahinter liegenden Zustandsmodellen.

- Alle weiteren Diagramme, wie z.B. Use-Case-, Objekt- und Paketdiagramme, wurden nur unwesentlich verändert.

3. Interaktionsdiagramme

Eines der Grundprinzipien der UML-Sprachschöpfer war es grundlegende Konzepte stärker in den Vordergrund zu stellen als Diagramme. Dies wird bei der Interaktionsmodellierung besonders deutlich. Es werden erst einmal nur grundlegende Konzepte definiert, die zur Darstellung von Kommunikationssituationen erforderlich sind. Dazu zählen als bedeutendste

- Interaktionen,
- Lebenslinien,
- Nachrichten,
- Kommunikationspartner und
- Sprachmittel zur Flusskontrolle.

Je nach Situation werden aber nicht alle UML-Elemente in ihrer kompletten „Notationstiefe“ benötigt. Aus diesem Grund wurden in der UML verschiedene Typen von Interaktionsdiagrammen definiert. Je nachdem ob nur die beteiligten Kommunikationspartner, die exakte zeitliche Reihenfolge oder Kommunikationsfehler und die daraus resultierenden Systemreaktionen modelliert werden sollen, stellt die UML verschiedene Diagrammtypen zur Verfügung. Dies sind die folgenden vier Diagramme:

- Sequenzdiagramm
- Kommunikationsdiagramm
- Timing-Diagramm
- Interaktionsübersichtsdiagramm

Es besteht ebenfalls die Möglichkeit im Rahmen der UML-Erweiterungsmechanismen eigene Diagramme zu entwerfen.

Abbildung 2 zeigt die drei möglichen Varianten zum darstellen von Lebenslinie und Nachricht am Beispiel eines Kochs der an einem Herd (Kommunikationspartner) die Temperatur einstellt (Nachricht).

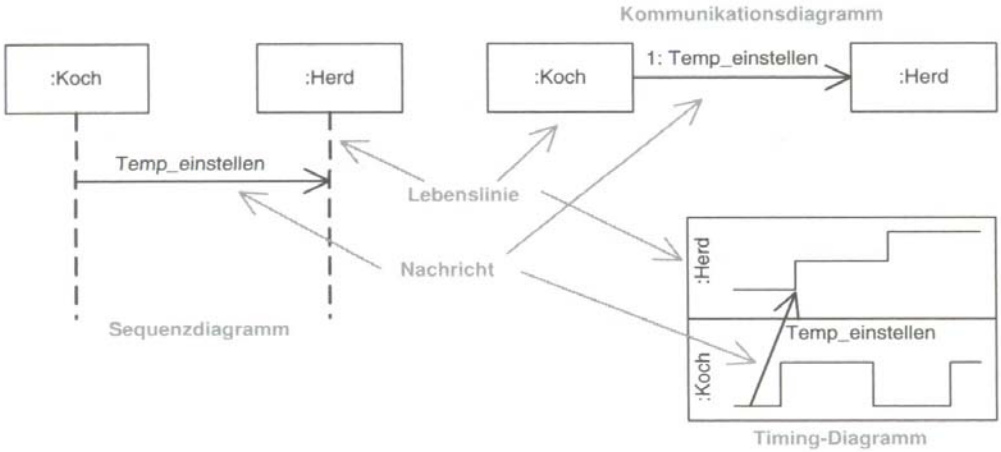


Abbildung 2: Grundelemente der Interaktionsmodellierung in ihren Diagrammvarianten [1]

4. Modellierung von Interaktionen

Von Interaktionen spricht man immer dann, wenn zwei oder mehre Einheiten miteinander kommunizieren. Die Grundelemente von Interaktionen sind die Lebenslinien, die die einzelnen Kommunikationspartner repräsentieren und die Nachrichten die zwischen den Kommunikationspartnern versendet werden. Bei der Nachricht kann es sich um verschiedenes handeln, z.B. der Aufruf oder die Rückantwort einer Operation (bei einer Klasse), ein Zeitsignal, das setzen einer Variablen oder ein logisches, analytisches Ereignis (wie: Käufer unterschreibt Vertrag). Das Grundprinzip einer Interaktion wird in Abbildung 3 dargestellt.

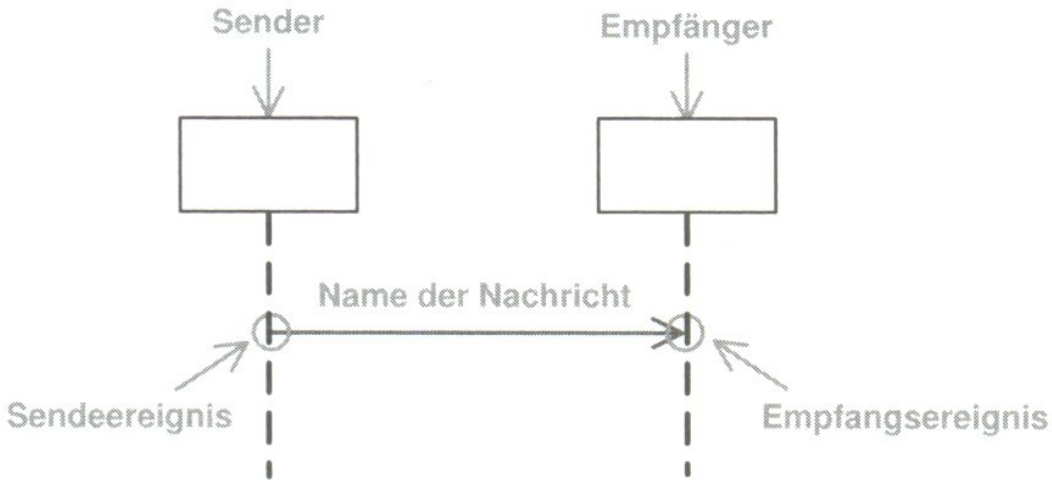


Abbildung 3: Grundprinzip einer Interaktion, hier in Sequenzdiagrammform [1]

5. Timing-Diagramme

Timing-Diagramme werden schon seit langem erfolgreich in der Elektrotechnik angewendet um das zeitliche Verhalten digitaler Schaltungen zu beschreiben. Seit Version 2.0 ist es nun auch in der UML verfügbar. Hauptaufgabe des Diagramms ist es zu zeigen wann sich verschiedene Interaktionspartner in welchem Zustand befinden.

5.1 Überblick

Das Timing-Diagramm ermöglicht die präzise Analyse und Darstellung des zeitlichen Verhaltens von Klassen, Akteuren, Komponenten, Schnittstellen usw. Dazu enthält es folgende Notationselemente:

- Lebenslinien und die dazu gehörigen Bedingungen, Zustände und Attributwerte,
- Nachrichten,
- Zeitbedingungen,
- Zeitverlaufslinien

Die Lebenslinien sind breite Streifen die die Zustände bzw. Bedingungen enthalten. Mehrere Lebenslinien werden vertikal angeordnet. Die Zeitachse wird horizontal angetragen. Sie wird üblicherweise mit einer Zeitskala versehen. Die Zeitverlaufslinie jeder Lebenslinie zeigt an, wann sich die Lebenslinie in welchem Zustand befindet. Waagrechte Linien beschreiben einen Zustand, senkrechte Linien einen Zustandswechsel. Nachrichten werden durch Pfeile zwischen den einzelnen Zeitverlaufslinien dargestellt. Abbildung 4 zeigt eine Übersicht über die häufigsten Notationselemente am Beispiel einer Fußgängerampel.

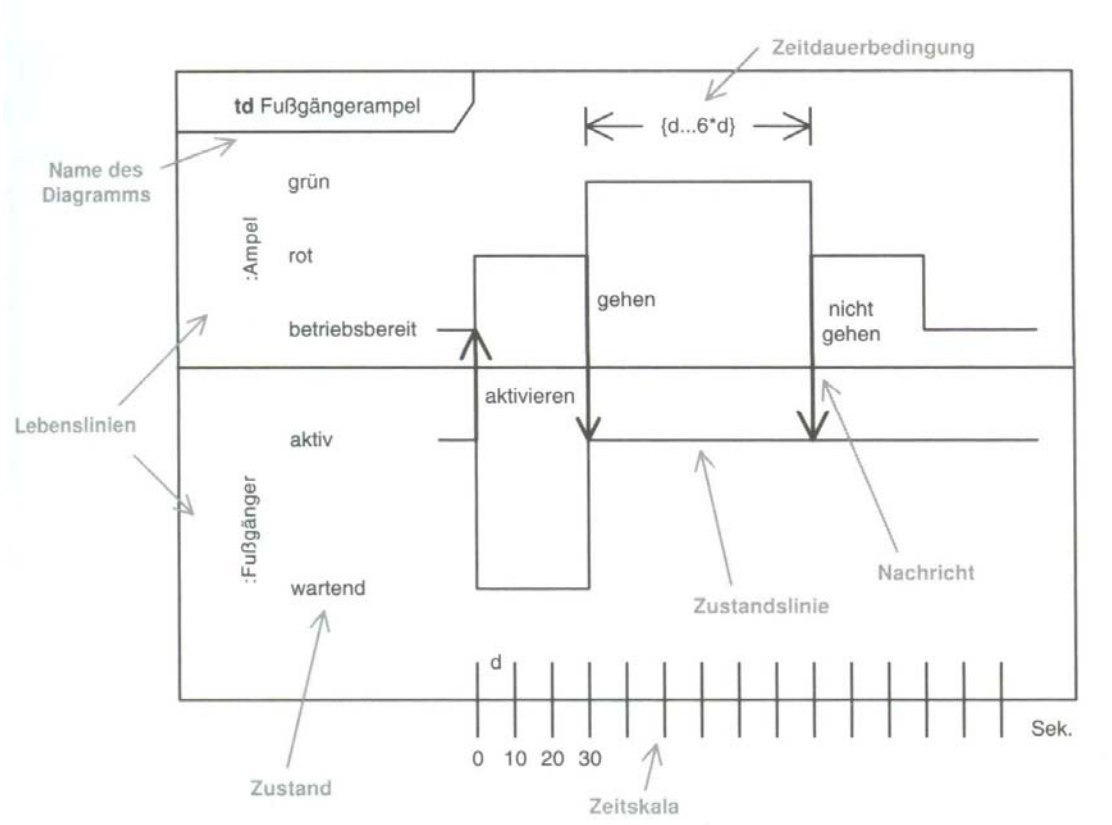


Abbildung 4: Das Timing-Diagramm und seiner Elemente [1]

Timing-Diagramme finden dann Anwendung wenn genaue zeitliche Übergänge wichtig sind, wenn lokale oder globale Daten für den Sachverhalt weniger interessant sind und wenn nicht mehrere Varianten auf einmal gezeigt werden sollen.

5.2 Anwendungsbeispiel

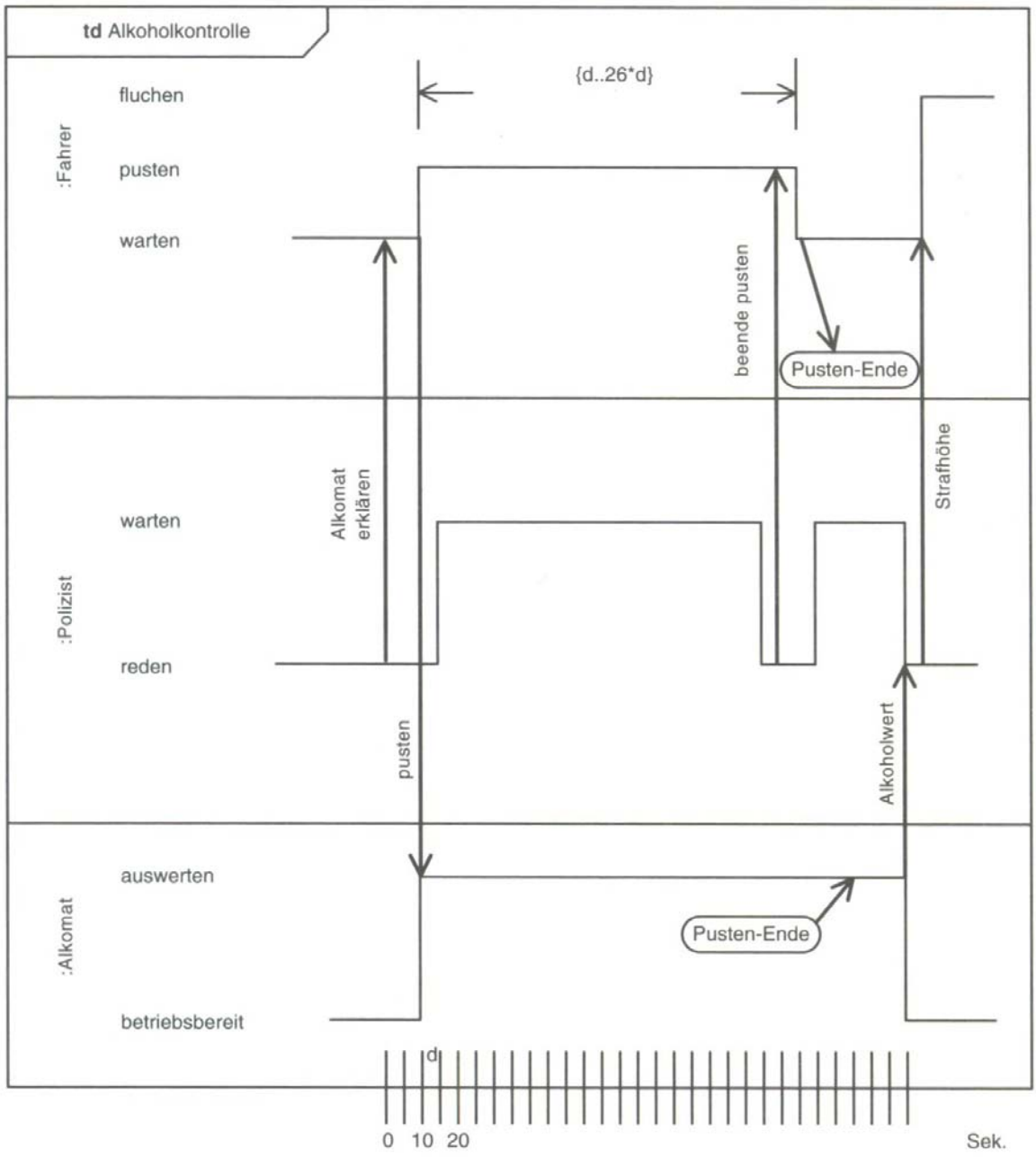


Abbildung 5: Der zeitliche Ablauf einer nächtlichen Alkoholkontrolle [1]

Abbildung 5 zeigt ein mögliches Szenario bei einer Alkoholkontrolle. Zunächst wird dem Fahrer die Bedienung des Alkomaten erklärt. Durch das pusten des Fahrers wechselt der Alkomat seinen Zustand und beginnt mit der Auswertung. Nach einer bestimmten Zeit fordert der Polizist den Fahrer auf das pusten zu beenden. Nach einer Wartezeit liefert der Alkomat dem Polizisten den Alkoholwert. Danach wird dem Fahrer die Höhe der Strafe mitgeteilt.

5.3 Anwendung im Projekt

Timing-Diagramme kommen vor allem dann zum Einsatz wenn das zeitliche Verhalten eines Systems eine entscheidende Rolle spielt. Es kann z.B. die Funktionalität einer elektronischen Schaltung oder eine vollautomatische Waschstraße modelliert werden. Insbesondere bei sicherheitskritischen Systemen, bei denen es auf den exakten Eintrittszeitpunkt von Ereignissen ankommt, muss das Zeitverhalten definiert werden. Timing-Diagramme beschränken sich dabei hauptsächlich auf Teile eines Systems, bei denen ein Fehler im zeitlichen Verhalten zu einem kritischen Fehlverhalten des kompletten Systems führt. Meist ist es aus technischer Sicht unmöglich das gesamte System mit Hilfe von Timing-Diagrammen zu modellieren.
Das Timing-Diagramm kann dabei immer nur den idealen Ablauf wiedergeben. Für Alternative oder Parallele Abläufe sieht die UML 2 keine Notationselemente vor.

5.4 Notationselemente

5.4.1 Interaktion / Interaktionsrahmen

Das Timing-Diagramm wird von einem rechteckigen Rahmen umgeben. Der Name der Interaktion steht in einem Fünfeck in der rechten oberen Ecke. Der Rahmen umschließt das komplette Diagramm. Wenn das Diagramm nur eine Komponente enthält ist der Rahmen gleichzeitig die Lebenslinie der Komponente. Für die Darstellung konkreter Zeitpunkte und Zeitdifferenzen kann auf der unteren Linie des Rahmen ein Zeitskala aufgetragen werden. Abbildung 6 zeigt einen Interaktionsrahmen mit Zeitskala.

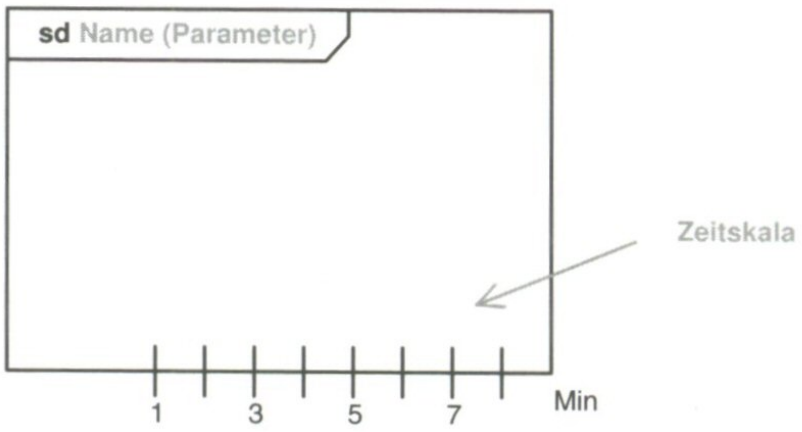


Abbildung 6: Interaktionsrahmen eines Timing-Diagramms mit Zeitskala [1]

5.4.2 Lebenslinie

Die Lebenslinie ist keine einfache Linie, sondern ein Bereich. Mehrere Lebenslinien werden durch durchgezogene horizontale Linien getrennt. (siehe Abbildung 7).

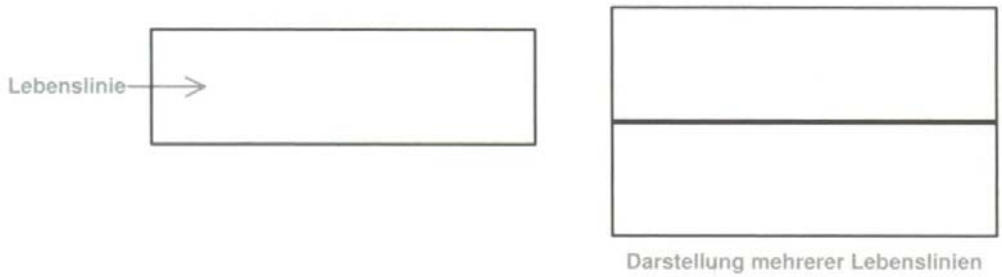


Abbildung 7: Notation von Lebenslinien in Timing-Diagrammen [1]

Jede Lebenslinie entspricht genau einem Teilnehmer einer Interaktion. Sie muss den Namen des Kommunikationspartners enthalten.

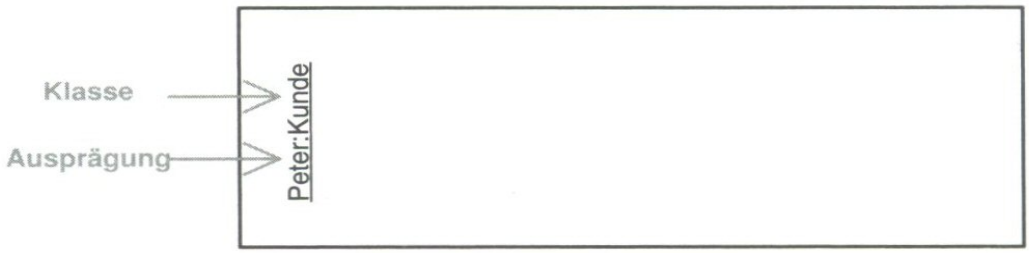


Abbildung 8: Namensgebung einer „Objekt“-Lebenslinie [1]

Abbildung 8 zeigt eine geläufige Benennung für eine Lebenslinie, die ein Objekt repräsentiert. Es kann auch nur dir Ausprägung oder die Klasse verwendet werden.

5.4.3 Zeitverlaufslinie

Änderungen von Zuständen, Attributen und gültigen Bedingungen einer Lebenslinie werden mit Hilfe einer Zeitverlaufslinie in Abhängigkeit von der Zeit oder speziellen Ereignissen dargestellt. Die Zustände einer Lebenslinie werden untereinander angeordnet (siehe Abbildung 9). Mit Hilfe der Zeitverlaufslinie wird die zeitliche Reihenfolge der Zustände angegeben. Eine vertikale Linie symbolisiert einen Zustandswechsel, eine horizontale Linie das verweilen in einem Zustand. Da in realen Systemen Zustandswechsel nicht zeitlos sind können Zustandswechsel auch durch schräg verlaufende Zeitverlaufslinien dargestellt werden

(siehe Abbildung 10). Der Auslöser für einen Zustandsübergang kann als Text an den entsprechenden Punkten der Zeitverlaufslinie vermerkt werden.

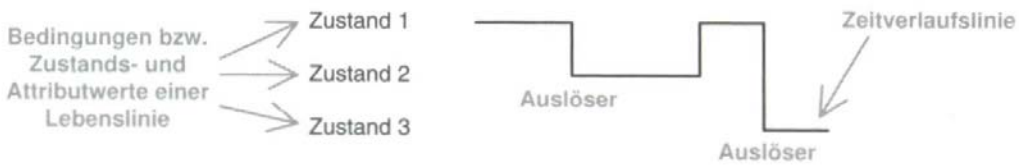


Abbildung 9: Zeitverlaufslinie [1]

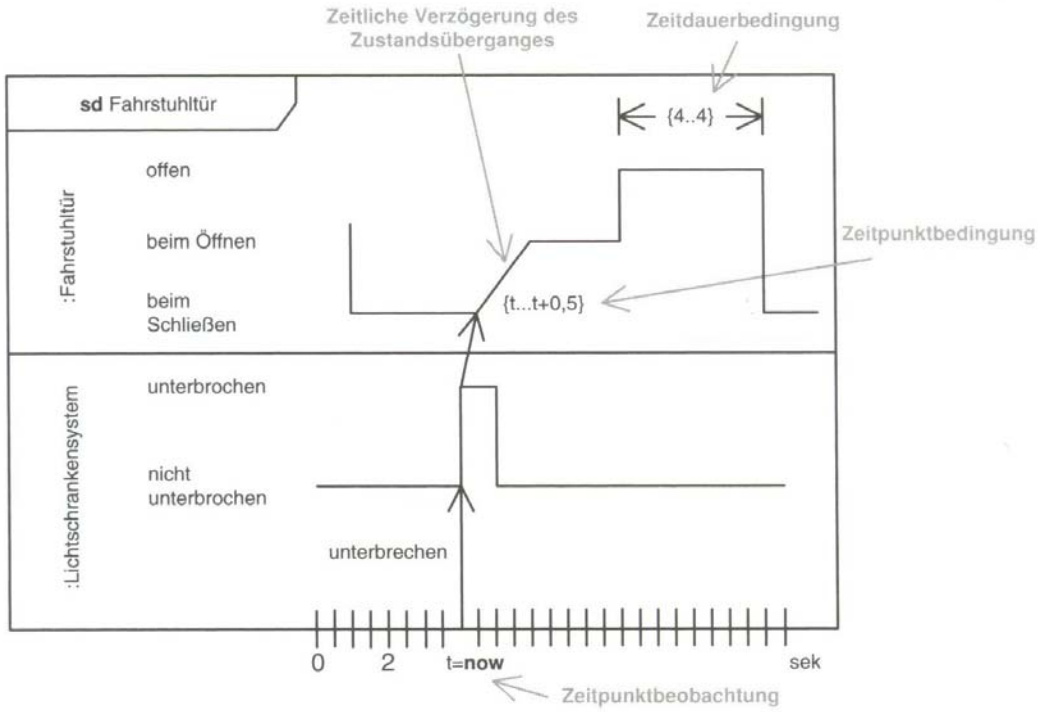


Abbildung 10: Zeitbedingungen einer Fahrstuhltür [1]

Mit Hilfe der Zeitskala kann die Verweildauer in einem Zustand genau angegeben werden. Das Ende einer Zeitverlaufslinie wird durch ein Stoppereignis modelliert (siehe Abbildung 11).

Abbildung 11: Stoppereignis einer Zeitverlaufslinie [1]

5.4.4 Nachricht

Der Informationsfluss zwischen den einzelnen Kommunikationspartner wird durch Nachrichten verkörpert. Eine Nachricht wird durch einen Pfeil zwischen den Zeitverlaufslinien dargestellt und ist immer vom Sender zum Empfänger gerichtet. Jede Nachricht muss benannt werden. Man unterscheidet zwischen synchronen und asynchronen Nachrichten. Bei synchroner Kommunikation wartet der Sender auf eine Antwortnachricht des Empfängers bevor er seine Abarbeitung fortsetzt. Nach dem senden einer asynchronen Nachricht setzt der Sender seiner Abarbeitung direkt fort. Die verschiedenen Typen können durch die Pfeilspitze unterschieden werden (siehe Abbildung 12). Asynchrone Nachrichten haben eine offene, synchrone eine ausgefüllte Pfeilspitze. Antwortnachrichten werden gestrichelt gezeichnet und haben eine ausgefüllte Spitze.

Abbildung 12: Notation von Nachrichten [1]

Da in Timing-Diagrammen der zeitliche Verlauf von Zustandsänderungen im Vordergrund steht interessieren hauptsächlich Nachrichten die solche Änderungen auslösen. Nachrichten die verzögert beim Empfänger ankommen werden durch schräge Pfeile dargestellt. Mit Hilfe der Zeitskala kann die Dauer der Nachrichtenübermittlung exakt bestimmt werden.

5.4.5 Sprungmarke

Sprungmarken verbessern die Lesbarkeit von Timing-Diagrammen mit vielen Lebenslinien. Sie unterbrechen eine Nachrichtenlinie, ohne dabei die Semantik der Interaktion zu beeinträchtigen. Sie werden hauptsächlich bei weit voneinander entfernten Lebenslinien eingesetzt. Das Timing-Diagramm ist das einzige Interaktionsdiagramm das Sprungmarken definiert.

Abbildung 13: Sprungmarken an Nachrichten [1]

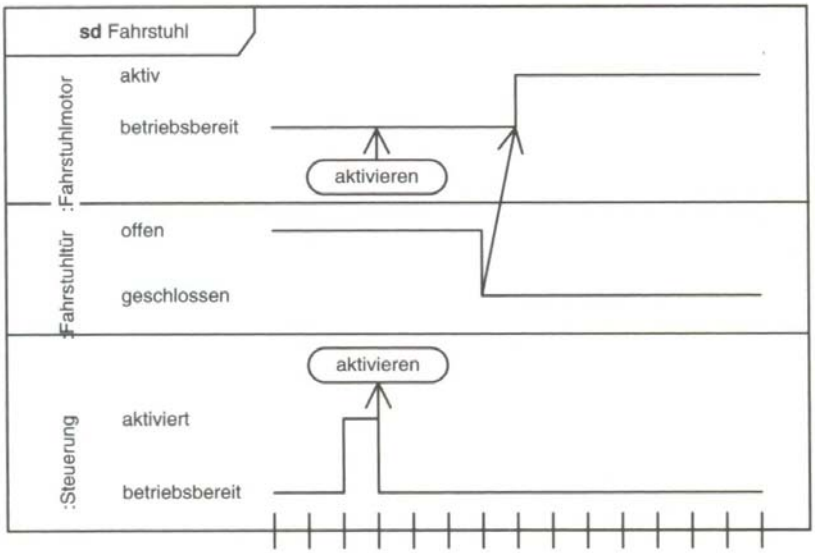


Abbildung 14: Verwendung von Sprungmarken [1]

5.4.6 Wertverlaufslinie

Die Wertverlaufslinie bietet eine alternative zur Zeitverlaufslinie. Sie kommt zum Einsatz wenn sehr viele Lebenslinien, viele Zustände oder Folge von Wertänderungen dargestellt werden sollen. Bei zu vielen Zuständen wird die Lebenslinie schnell unübersichtlich. Da die Zeitverlaufslinie immer nur genau einen Zustand einnehmen kann, können die Zustände auch horizontal angeordnet werden. Die Zeit die eine Lebenslinie in einem Zustand verweilt wird durch die Breite einer Wabe symbolisiert. Zustandsänderungen werden durch die Kreuzungspunkte dargestellt.

Abbildung 15: Notation der Wertverlaufslinie [1]

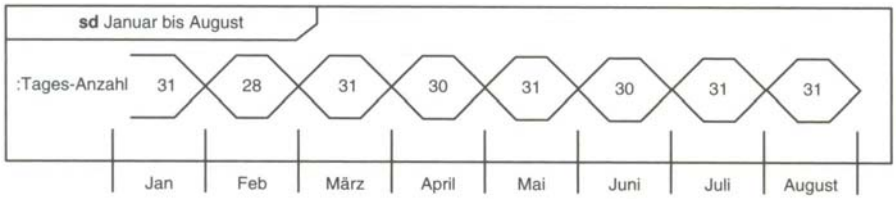


Abbildung 16: Tageszahlen eines Monats [1]

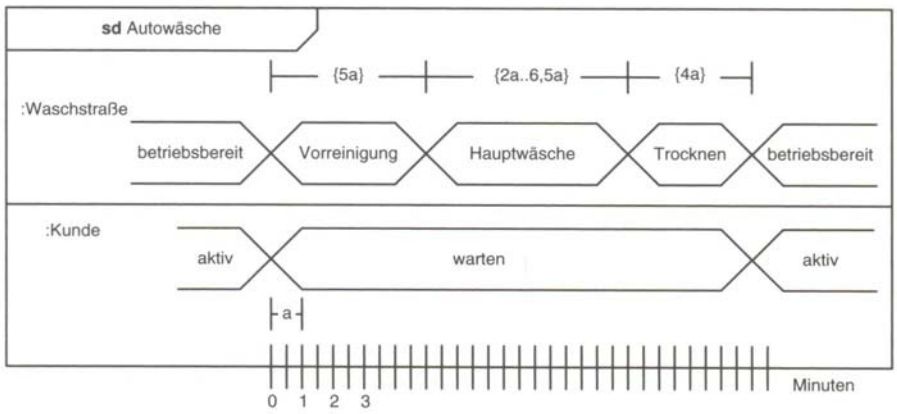


Abbildung 17: Eine Waschstraße mit einer Wertverlaufslinie [1]

6. Fazit

Die neueste Version der UML ist um einiges kompakter geworden als ihre Vorgänger. Die grundlegenden Konzepte der Infrastruktur wurden angepasst, womit ein wichtiger Schritt in Richtung Wiederverwendung und Straffung der Konzepte getan wurde. Leider ist die neue UML 2 nicht vollständig Abwärtskompatibel. Durch das veränderte Aussehen einiger grafischer Notationselemente kann es bei Umsteigern anfänglich zu Verwechslungen kommen. Auch einige eingeführte Bezeichnungen und Namen wurden geändert, was ebenfalls für Benutzer früherer Versionen verwirrend sein kann.
Mit Hilfe der in der UML 2 neu eingeführten Timing-Diagramme ist es nun möglich auch zeitkritische Systeme wie z.B. ABS-Systeme oder Atomkraftwerke zu modellieren.

7. Literaturverzeichnis

[1] Jeckle, Rupp, Hahn, Zengler, Queins; UML 2 glasklar; Hanser Verlag; 2004

[2] www.uml-glasklar.de

[3] Fowler, Martin; UML Distilled Third Edition; Addison-Wesley; 2004

[4] www.uml.org

[5] Oestereich, Bernd; www.oose.de/uml

[6] UML 2.0 Infrastructure Final Adopted Specifcation; www.omg.org/technology/documents/modeling_spec_catalog.htm#UML

[7] UML 2.0 Superstructure Final Adopted specification; www.omg.org/technology/documents/modeling_spec_catalog.htm#UML

[8] Jeckle, Rupp, Hahn, Zengler, Queins; UML 2.0: Evolution oder Degeneration?; www.jeckle.de/UML2Artikel-OS/index.html

8. Abbildungen

Abb. 1: Die Diagrammtypen der UML 2
Abb. 2: Grundelemente der Interaktionsmodellierung in ihren Diagrammvarianten
Abb. 3: Grundprinzip einer Interaktion, hier in Sequenzdiagrammform
Abb. 4: Das Timing-Diagramm und seiner Elemente
Abb. 5: Der zeitliche Ablauf einer nächtlichen Alkoholkontrolle
Abb. 6: Interaktionsrahmen eines Timing-Diagramms mit Zeitskala
Abb. 7: Notation von Lebenslinien in Timing-Diagrammen
Abb. 8: Namensgebung einer „Objekt“-Lebenslinie
Abb. 9: Zeitverlaufslinie
Abb. 10: Zeitbedingungen einer Fahrstuhltür
Abb. 11: Stoppereignis einer Zeitverlaufslinie
Abb. 12: Notation von Nachrichten
Abb. 13: Sprungmarken an Nachrichten
Abb. 14: Verwendung von Sprungmarken
Abb. 15: Notation der Wertverlaufslinie
Abb. 16: Tageszahlen eines Monats
Abb. 17: Eine Waschstraße mit einer Wertverlaufslinie

UML 2.0 Quelltextgenerierung

Seminararbeit
im Fach Informatik
im Rahmen des Seminars "**Sicherheitskritische Systeme**"
an der

Universität Siegen, Fachgruppe für Praktische Informatik

eingereicht bei
Dr. Jörg Niere

vorgelegt von
Beyhan Goelgeli

Sommersemester 2004

1 EINLEITUNG

2 VERGLEICH DES UML 2.0 UND DES VORGÄNGERVERSIONS 1.X

3 QUELLTEXTGENERIERUNG AUS DEN UML-DIAGRAMMEN

3.1 Model-Driven-Architecture (MDA)

3.2 MOF

3.3 PIM(Plattform Independent Model)

3.4 PSM(Plattform Specific Model)

3.5 Templates

3.6 Abbildung der Modelle(Mapping)
3.6.1 PIM nach PIM
3.6.2 PIM nach PSM
3.6.3 PSM nach PSM
3.6.4 PSM nach PIM

1 Einleitung

Die UML enthält die Konventionen, mit denen Softwaredesigner und -Entwickler Softwareprojekte eindeutig spezifizieren, modellieren und dokumentieren können. Mit UML 2.0 eingeführte Erweiterungen und Verbesserungen lassen sich noch besser zur Modellierung komponentenbasierter Systeme einsetzen. Das Hauptziel bei der Definition von UML 2.0 war die sog. MDA(Model Driven Architecture). Mit diesem Konzept lassen sich mit den speziellen Softwarewerkzeugen ausführbare Programme theoretisch direkt aus den Modellen erzeugen. Dazu müssten aber Compiler bzw. Interpreter für die jeweilige Plattformen existieren, die plattformspezifischen Modelle ausführbar zu machen. Solche Compiler oder Interpreter existieren nur in Nischenbereichen bzw. in der Diskussion.

2 Vergleich des UML 2.0 und des Vorgängerversions 1.x

Mit UML 2.0 ist es Möglich 14 verschiedene Diagramme zu erfassen, die verschiedene Aspekte des Systems modellieren. Überwiegend alle Diagrammarten wurden überarbeitet und mit neuen Elementen ausgestattet.

- Strukturdiagramme
    - Klassendiagramme
    - Kompositionsstrukturdiagramme (neu)
    - Komponentendiagramme
    - Paketdiagramme
    - Verteilungsdiagramme
    - Objektdiagramme
- Verhaltensdiagramme
    - Aktivitätsdiagramme
    - Zustandsautomat Diagramme
    - Anwendungsfalldiagramme
    - Interaktionsdiagramme
        - Kommunikationsdiagramme
        - Interaktionsübersichtsdiagramme(neu)
        - Sequenzdiagramme
        - Timingdiagramme (neu)

Das Kompositionsstrukturdiagramm

Diese Diagrammart wurde bei UML 2.0 neu eingeführt und dient der Darstellung der internen Struktur eines Klassifiziers und seinen Beziehungen zu anderen Systemteilen. Diese Diagrammart beschäftigt sich mit der Struktur der einzelnen Architekturkomponenten und mit deren Zusammenhang zwischen den Architekturkomponenten. Dieses Diagramm dient hauptsächlich zur abstrakten Dokumentation des Zusammenwirkens der einzelnen Architekturkomponenten. In einem Top-Down Prozess können die Bestandteile des Systems verfeinert werden.

Das Interaktionsübersichtsdiagramm

In dem Interaktionsübersichtdiagrammen wird eine Menge von Interaktionen in einer höheren Abstraktionsebene dargestellt. Diese Diagrammart ist für kleine und mittelgroße Projekte sinnvoll, da mit zunehmendem Komplexitätsgrad der Softwaresysteme, ziemlich aufwendig ist, diese Diagramme zu erstellen. In dem Interaktionsübersichtsdiagrammen werden die Kontrollmechanismen des Aktivitätsdiagramms genutzt.

Das Timingdiagramm

Mit den Timingdiagrammen werden unter Angabe der exakten zeitlichen Bedingungen die Zustandsänderungen eines Klassifiziers beschrieben. Diese Diagrammart ist vor allem in der Digitaltechnik sehr sinnvoll.

Das Aktivitätsdiagramm

Die Spezifikation der Aktivitätsdiagramme wurde komplett überarbeitet. Der Semantik dieser Diagrammart an die Petri-Netze angelehnt. Es sind folgende ergänzende Elemente hinzugefügt: strukturierte Knoten, Entscheidungsknoten, Schleifenknoten, Mengenverarbeitungsknoten etc.

Das Aktivitätsdiagramm

Die Aktivitätsdiagramme können Vor- und Nachbedingungen haben und die Aktivitäten können Ein- und Ausgaben haben. Das Verhalten an Verzeigungs- und Verbindungsknoten können präziser beschrieben werden. Es dürfen mehrere Startknoten geben, die dann zu parallelen Abläufen führen.

3 Quelltextgenerierung aus den UML-Diagrammen

Die Codegenerierung aus dem UML 2.0 Diagrammen basiert auf Model Driven Architecture (MDA). MDA ist ein ziemlich neuer Standard von OMG. Mit MDA soll eine erhebliche Steigerung der Entwicklungsprozess erreicht werden. Aus formal eindeutigen Modellen können durch Generatoren automatisch Code erzeugt werden. Durch den Einsatz der Generatoren und die formal eindeutig definierten Modellierungssprachen wird auch darüber hinaus die Softwarequalität gesteigert. Die Qualität des automatisch erzeugten Quellcode ist gleich bleibend, was zu einem höheren Grad der Wiederverwendung führt.

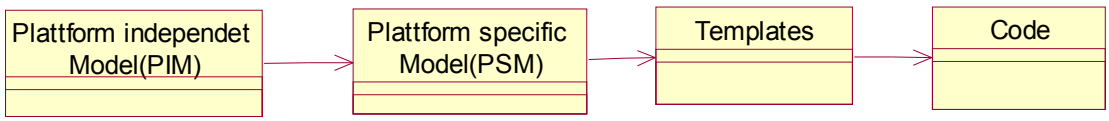


Abbildung 1: Die Stufen der Codegenerierung

Die Codegenerierung von den UML-Diagrammen läuft in unterschiedlichen Stufen ab[siehe Abbildung 1]. Einer von wichtigsten Konzepten von MDA ist, dass die Idee von multiplen Ebenen von Modellen. Die Codegenerierung fängt bei der höchsten Ebene, genannt PIM an. Die Modelle auf dieser Ebene sind mit einer plattformunabhängigen Sprache spezifiziert. Die Spezifikationen von PIM ist daher sehr abstrakt gehalten, so dass alle möglichen software-technischen Anforderungen erfassbar sind. Der nächste darunter liegende Ebene ist plattform-spezifisch, kurz PSM genannt. Diese Ebene PSM erweitert das PIM mit den notwendigen Strukturen, so dass die Generatoren daraus Quellcode erzeugen können. Die Pfeile zwischen den Stufen bei Abbildung 1 bilden die Transformationen, die durch Transformationsregeln geregelt werden.

Die Transformation zwischen plattform-spezifischen Modell und Templates ist der entscheidende Teil für die Codegenerierung des MDA-Modells. Die Ebene zwischen das plattform-abhängigen Modell

und das Template führt alle notwendigen Transformationen zwischen PSM und die Implementierungsspezifikation eines Systems durch. Das bedeutet, dass die Templates nur noch den Quellcode erzeugen müssen, nicht die in PSM spezifizierten Modelle interpretieren und anschließend den Code erzeugen müssen. Das MDA- Konzept entspricht in vereinfachter Form dem 3-Tier Modell(Also Benutzerschicht, Fachkonzeptschicht und Datenhaltungsschicht).

3.1 Model-Driven-Architecture (MDA)

Die Heutige Softwareentwicklung ist davon geprägt, dass die Softwareentwickler während der Anforderungsanalyse und Modellierung von Geschäftsprozessen noch unabhängig von der Zielplattform arbeiten können.

Die MDA ist ein Schlüssel-Konzept von OMG zur Modellierung von u.a. Softwaresystemen, welche durch eine klare Trennung von Abstraktionsschichten die Wiederverwendbarkeit und Langlebigkeit der Modelle ermöglicht. Einer der wesentlichen Ziele von MDA ist es die Technologien wie CORBA, XML, J2EE und .Net zu integrieren, um so die Portabilität von Software auf verschiedenen Plattformen unterstützen zu können. Mit der MDA wird versucht die Lücke zwischen Modellen und Quelletexten zu schließen. Die MDA beschreibt Modelle mit Hilfe der UML und unterscheidet zwischen zwei wichtigen Modellen bezüglich ihrem Abstraktionsgrad PIM oder PSM. Die PIM-Modelle enthalten formale Spezifikationen der Strukturen und Funktionalitäten des Systems, die von technischen Details abstrahieren. Die PSM-Modelle werden aus dem PIM anhand vorgegebener Transformationsregeln und –schritte abgeleitet.

Ein weiteres zentrales Element von MDA ist das Mappen zwischen Modellen. Die MDA ermöglicht Modelle zwischen den verschiedenen Geschäftsbereichen oder zwischen den Modellebenen aufeinander abzubilden(engl. Mapping). Das Mappen kann je nach Realisierungsgrad des MDA-Konzeptes automatisch verlaufen. Die Abbildung 2 stellt eine Metamodell-Beschreibung der MDA dar. MDA wird von verschiedenen Case-Werkzeugen u.a. Rational Software von Rational Rose, Visio von Microsoft und Poseidon von Gentleware unterstützt.

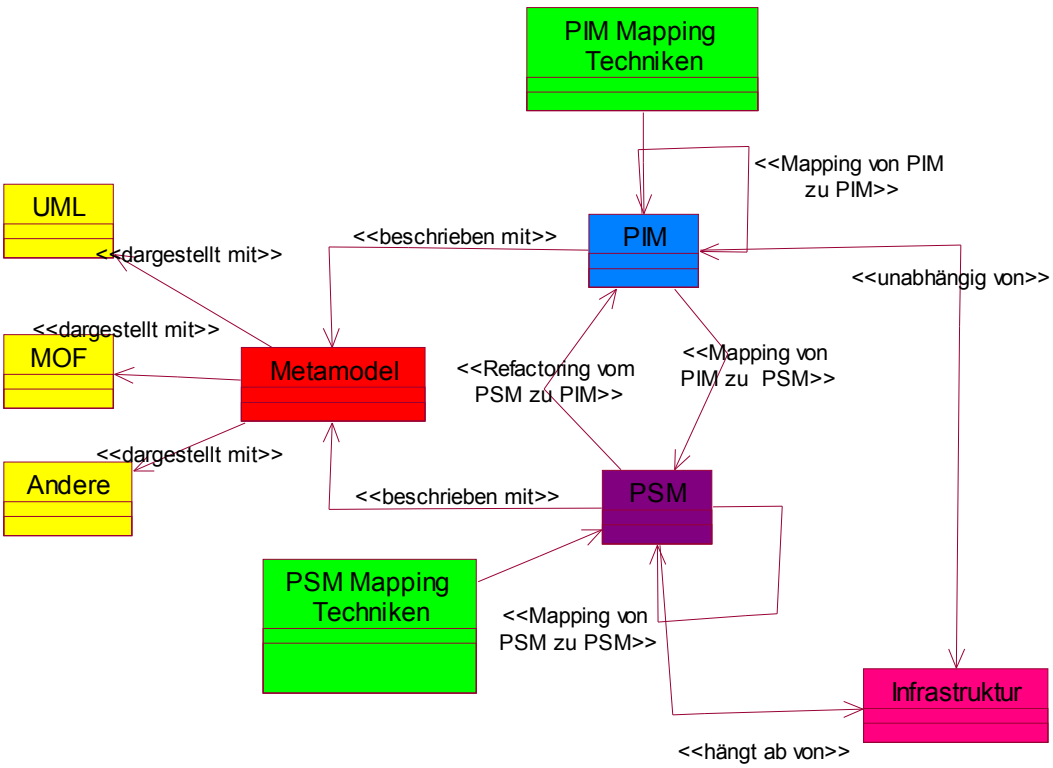


Abbildung 2: detaillierte Beschreibung der MDA

3.2 MOF

Dadurch sollen sich Modelle und Metamodelle besser zwischen verschiedenen Werkzeugen austauschen lassen. Ziel ist es, Entwicklern und Designer eine größere Freiheit bei der Auswahl der verwendeten Tools zu ermöglichen.

Die Meta Object Facility (MOF) ist eine abstrakte Beschreibungssprache, die zur Erstellung und Verwaltung von plattform-unabhängigen Metamodellen dient. Beispiele solcher Metamodelle sind die UML und MOF selbst. Dadurch sollen sich Modelle und Metamodelle besser zwischen verschiedenen Tools austauschen lassen. Ziel ist es dabei den Entwicklern und Designern eine größere Freiheit bei der Auswahl der verwendeten Tools zu ermöglichen. Es gibt verschiedene Arten von Modellen, die verschiedene Aspekte der Softwaresysteme beschreiben. Um die Portabilität dieser Modelle untereinander zu gewährleisten, braucht man eine eindeutige und einheitliche Spezifikationen um diese Modellierungssprachen zu beschreiben. Die MOF umfasst all diese standardisierte Spezifikationen. Mit MOF ist möglich alle andere Art von Modellen zu beschreiben. Die MOF reduziert den Transformationsaufwand der Modelle untereinander.

Ohne abstrakte Beschreibungssprache MOF müsste man (siehe Abbildung 3) pro Modell mindestens 3 Transformationen durchführen. Mit MOF als Beschreibungssprache ist nur eine MOF-Mapping für eine Modellierungssprache notwendig.

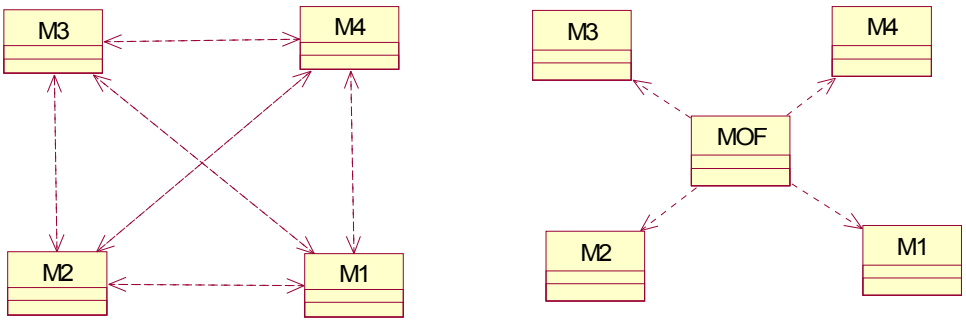


Abbildung 3: Mapping der Modelle: links ohne MOF, rechts mit MOF

Die Metamodellarchitektur

Die Architektur von MOF besteht aus sog. 4-Meta-Ebenen, nämlich M0, M1, M2 und M3(siehe Abbildung 4)

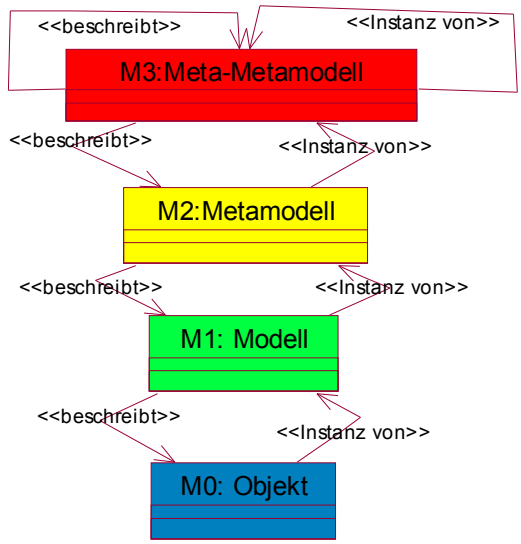


Abbildung 4: 4-Schichten Architektur von MOF

- Auf der Ebene M0(auch als User-ebene bezeichnet) werden konkrete Datenkonfigurationen eines Systems dargestellt. Bezogen auf ein Programm wäre dies: der Ausführungszustand des Programms
- Die Ebene M1 beschreibt die Sprache zur Beschreibung von Domänen.(z.B. Klasse:Schüler, Klasse:Buch…). Auf dieser Ebene werden die möglichen Zustände eines Systems beschrieben. Diese Ebene wird auch als Metadaten Modell, die Modellebene genannt. Die Objekte aus der M0 sind die Instanzen aus der Ebene M1. Diese Ebene ist vergleichbar mit einem Quellcode eines Programms.
- Auf der Ebene M2(wird auch als Metamodell-Ebene genannt)werden Elemente spezifiziert, die zur Modellierung auf der Ebene M1 eingesetzt werden. Auf dieser Ebene werden die Modelle mit zusätzlichen Elementen wie Variable, Operationen ergänzt. Diese Schicht ist das zentrale Element der UML. Auf dieser Ebene beschriebenen Konzepte werden bei der UML-Modellierung verwendet.(wie Z.B.: Die Klassen, Attribute, Assoziationen…) Durch Diese MetaModell ist eine abstrakte Syntax der Modellierungssprache definiert. Diese Ebene würde mit EBNF einer Programmiersprache vergleichbar. Alle Elemente, die auf dieser Ebene zur Definition des Metamodells verwendet werden, wird auf der Ebene M3, Meta-Metamodell-Ebene spezifiziert.
- Die Ebene M3 bildet die Infrastruktur der Metamodellarchitektur. Dieser Schicht ist für alle Modellierungssprachen, die mit MOF definiert werden, ist identisch. Dieser Schicht wird auch MOF-Ebene genannt, weil die Regeln um MOF selbst zu beschreiben, auf dieser Ebene beschrieben werden. Zur Vervollständigung der Analogie wäre dieser Ebene die Definition der Syntax des EBNF.

3.3 PIM(Plattform Independent Model)

UML Modelle sind deklarative Modelle, wie auf IDL-basierende Objektmodelle. Die UML-Modelle können sowohl graphisch als auch textuell ausgedrückt werden. Die PIM-Modelle beschreiben Geschäftsverhalten und –funktionalitäten eines Systems völlig unabhängig von den technischen Details bzw. Eigenschaften der zu Grunde liegenden Plattform.

Die Modelle von verschiedenen Systemen werden in einem plattform-unabhängigen Modell

3.4 PSM(Plattform Specific Model)

Plattformspezifische Modelle werden genau so wie plattformunabhängige Modelle meistens mit UML spezifiziert. Bei denn PSM-Modellen haben die UML-Profile eine besondere Rolle. Sie ermöglichen dem Softwaredesigner plattformspezifische Eigenschaften in das Modell einzubinden. Für die spezifischen Technologien wurden bereits einige Profile definiert. Z.B. UML-Profil für CORBA, UML-Profil für JAVA.

3.5 Templates

Die Mapping beinhaltet Templates. Die Templates sind parametrisierte Modelle, die detailliert die Transformationsregeln spezifizieren. Die Templates sind mit Entwurfsmustern vergleichbar, aber sie enthalten plattform-spezifische Informationen um die Transformationen der Modelle durchzuführen. Die Templates können auch benutzt werden von einem Entwurfsmuster in einem anderen Entwurfsmuster zu transformieren.

3.6 Abbildung der Modelle(Mapping)

Durch die Mapping werden Modelle ineinander transformiert. Die Transformation läuft strikt nach den festgelegten und exakt spezifizierten Transformationsregeln ab. Die Transformationsregeln enthalten Informationen, wie die Modelle in ein anderes Modell

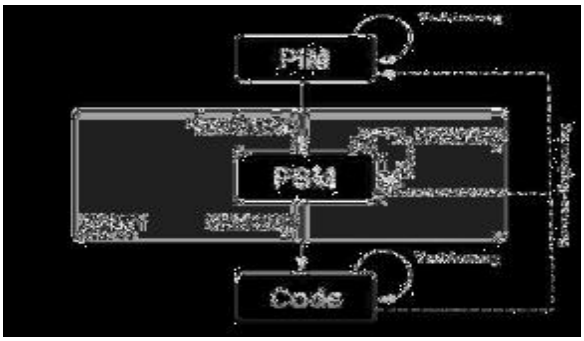


Abbildung 5: Übersicht der Transformationsvarianten

3.6.1 PIM nach PIM

Diese Transformation wird benutzt wenn die Modelle während dem Entwicklungsprozess erweitert, gefiltert, oder verfeinert werden, ohne plattform-abhängige Informationen zu nutzen. Die PIM zu PIM Mappings werden meistens eingesetzt, wenn die Modelle verfeinert werden müssen.

3.6.2 PIM nach PSM

Die Transformation von plattformunabhängigen Modellen an die speziellen Gegebenheiten einer technologie-abhängigen Plattform spielt eine zentrale Rolle in dem MDA-Konzept. Die Transformation von PIM nach PSM verläuft automatisch. Die Trennung von PIM und PSM Modelle bewirkt die Modelle einfacher zu verstehen, denn die Aspekte der beiden Modelle werden nicht vermischt. Diese Transformation ist essentiell wichtig für die Codegenerierung, weil zwei verschiedene Modelle aufeinander abgebildet werden, die besondere Transformationsregeln erfordern

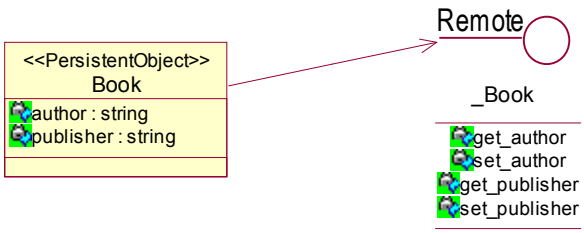


Abbildung 5: Transformation von PIM in das PSM

Falls für die angestrebte Zielplattform ein Codegenerator existiert, der die Transformation durchführt, kann wie bei Abbildung 5, kann PIM in ein PSM Modell umgewandelt werden. Um die automatische Codegenerierung zu unterstützen können auch Markierungen ins PIM eingefügt werden, die in einem UML-Profil definiert sind. Anhand dieser Markierungen kann der Codegenerator die Umwandlung präziser durchführen.

Durch die Nähe der PSM-Modelle zur Implementierungsplattform kann mit Hilfe der Templates relativ leicht und automatisch in Quellcode der Implementierungssprache übersetzt werden.

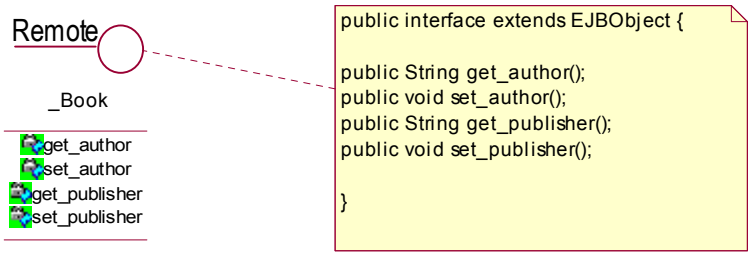


Abbildung 6: Codegenerierung aus dem PSM

3.6.3 PSM nach PSM

Die PSM-Modelle können auch bei Bedarf weiter verfeinert werden, um die fehlende Implementierungsdetails zu ergänzen, oder auch in ein anderes Zielplattform zu Transformieren. Der Umfang der erforderlichen Änderungen an Modellen ist von der Performance des benutzten Generators abhängig.

3.6.4 PSM nach PIM

Diese Transformation wird meistens bei bestehenden Systemen eingesetzt, um aus dem PSM-Modell ein PIM-Modell zu erzeugen. Die Transformation von PSM ins PIM ermöglicht die Abstraktion von Modellen der existierenden, plattformspezifischen Implementierungen zu plattformunabhängigen Modellen. Der Transformationsprozess ist allerdings recht-schwierig, da die Informationen über Metamodelle fehlen, es existieren CASE-Werkzeuge wie Rational Software von Rational Rose die Reverse Engineering(PSM nach PIM) ermöglichen, wenn auch ziemlich viele Randbedingungen innerhalb der automatisch generierten Quellcode beachtet werden müssen. Das neue transformierte Modell kann auch auf andere Plattformen portiert werden!

Glossar

Model Driven Architecture (MDA)	Die MDA(modell-getriebene Architektur) ist ein Konzept von OMG, die die funktionale Spezifikationen eines Systems von der Implementierungsspezifikationen trennt.
Plattform Independent Model(PIM)	Ein Model von Subsystemen, das keinerlei Informationen über spezifische Plattformen oder Technologien enthält. Ein technologieunabhängiges Modell eines Softwaresystems
Plattform Specific Model (PSM)	Ein PSM ist ein implementierungsnahes Modell eines Softwaresystems, das aus dem PIM generiert wird. Im MDA bildet PSM ist die Vorstufe zur Codegenerierung.
Unified Modeling Language(UML)	Eine Sammlung von mehreren Modellierungssprachen, die überwiegend für die Modellierung von objektorientierter Softwaresystemen eingesetzt wird.
Model	Ein Model ist eine abstrakte Repräsentation von Teil der Funktionalität, Struktur und/oder das Verhalten eines Systems.
Mapping	Eine Reihe von Regeln und Techniken, die benutzt werden um ein Model zu ändern oder in ein anderes Model zu transformieren.
Meta Object Facilities (MOF)	MOF ist eine Beschreibungssprache für alle Modellierungssprache der OMG. Mit MOF standardisierte Metamodelle und die Konzepte werden benutzt um andere Metamodelle zu definieren.
Metamodel	Ein Model, das eine Modellierungssprache definiert, mit dem einen Model im UML definiert wird. Die UML-Modelle basieren auf Meta-Modelle.(Model von Modellen)
InfraStuktur	Definiert für UML: UML-Profile, Stereotypen…
UML-Profile	UML-Profile sind der Standardmechanismus zur Erweiterung des Sprachumfangs der UML. Sie enthalten Sprachkonzepte, die über Basis-UML-Konstrukte wie Klassen und Assoziationen, Stereotypen und Modellierungsregeln festgelegt werden.
Interface Definition Language(IDL)	Wird verwendet die Schnittstellen vor allem bei CORBA, JAVA o.ä. zu definieren.

Referenzen

[1] http://www.codegeneration.net
[2] http://www.klasse.nl/english/mda/mda-introduction.html
[3] http://www.omg.org./technology/documents/formal/mof.htm
[4] http://www.omg.org/mda
[5] http://gentleware.com/products/documentation/poseidon_users_guide/generate.html#codeGen

Real Time Linux: Ein Überblick

Seminararbeit
im Fach Informatik
im Rahmen des Seminars "**Sicherheitskritische Systeme**"
an der

Universität Siegen, Fachgruppe für Praktische Informatik

eingereicht bei
Dr. Jörg Niere

vorgelegt von
Marcus Klein

Sommersemester 2004

Inhaltsverzeichnis

1 Einleitung

2 Warum ist nicht jedes Betriebssystem echtzeitfähig?
- 2.1 Aufgaben eines Betriebssystems
- 2.2 Optimierungen konventioneller Betriebssysteme
- 2.3 Anforderungen für ein echtzeitfähiges Linux
- 2.4 Scheduling unter Linux

3 Wie kann man Realzeitbetriebssysteme implementieren?
- 3.1 Welche Projekte implementieren Echtzeitfähigkeit für Linux?
- 3.2 KURT
  - 3.2.1 Verfeinerung der zeitlichen Auflösung mit UTIME
  - 3.2.2 Leistungsfähigkeit von UTIME
  - 3.2.3 Realzeitscheduling mit KURT
  - 3.2.4 Performance von KURT

4 Ausblick

Abbildungsverzeichnis

1 Laufzeitvergleich der Timerinterrupt Service Routinen
2 Abweichung der Softwareuhr
3 Einhaltung einer Schleifenzeit von 10ms
4 10ms-Schleife nach der Verbesserung
5 Zeitmessung vom Interrupt zur eingeplanten Routine
6 Zeitmessung vom Interrupt zur eingeplanten Routine (Verbesserung)
7 Dauer der Sperrung von Interrupts
8 Effektivität des Scheduler SCHED_FIFO
9 Effektivität des Scheduler SCHED_KURT
10 Effektivität der Scheduler im Vergleich
11 SCHED_ALL_PROCS und SCHED_KURT_PROCS im Vergleich
12 Zeiten mit Hintergrund-I/O-Aktivität

Tabellenverzeichnis

Listings

1 Pseudo-Code für die Linux Timer Interrupt Service Routine
2 Pseudo-Code für die UTIME Linux Timer Interrupt Service Routine

1 Einleitung

In Sicherheitskritischen Systemen werden zunehmend Rechner mit flexibel änderbarer Software eingesetzt, um andere starre, meist in Hardware realisierte Systeme zu ersetzen. Die Anforderung an den Rechner ist dabei in Echtzeit auf Ereignisse zu reagieren. Soll dies mit einem Standardpersonalcomputer (Standard-PC) realisiert werden, muß die Software die Einhaltung der Anforderungen gewährleisten. Man spricht dann von einem Echtzeitsystem.

Was bedeutet der Begriff Echtzeitsystem?

Ein Echtzeitsystem ist ein System, das ständig anfallende Daten sofort verarbeiten kann, wobei die Daten zufällig oder zu bestimmten Zeitpunkten eintreffen. Unter „sofort verarbeiten“ versteht man dabei, daß zu einem bestimmten Zeitpunkt die Abarbeitung beendet sein muß, so daß das Ergebnis der Abarbeitung zu diesem Zeitpunkt verwendet werden kann. Kann das System die Abarbeitung zu einem bestimmten Zeitpunkt nicht einhalten, so unterscheidet man zwischen zwei Fällen.

Weiche und harte Echtzeitfähigkeit

Weiche und harte Echtzeitfähigkeit werden anhand des Schadens unterschieden, der bei der Überschreitung der Zeitvorgaben auftritt. Handelt es sich dabei um einen kleinen Schaden oder gar nur einen störenden Effekt so spricht man von weichen Echtzeitanforderungen. Diese werden z.B. an die Wiedergabe von digital gespeicherten Video- und Audioinformationen gestellt. Tritt jedoch bei Verletzung der Zeitvorgaben sofort der maximale Schaden ein, so liegen harte Echtzeitanforderungen an das System vor. Dazu gehört z.B. das Auslösen des Airbags im Auto. Wird er nicht rechtzeitig gezündet, so prallt das Gesicht auf das Lenkrad.

Um Software für Echtzeitsysteme auf unterschiedlichen Rechnern komfortable einsetzen zu können, ist ein Betriebssystem notwendig, das zwischen Hardware und Software ein komfortable Schnittstelle zur Verfügung stellt. Diejenigen Aspekte von Betriebssystem, die bei Echtzeitsystemen eine Rolle spielen, werden im folgenden beleuchtet. Danach wird eine konkrete Umsetzung betrachtet, die das Betriebssystem Linux an die Anforderungen an Echtzeitsysteme anpaßt.

2 Warum ist nicht jedes Betriebssystem echtzeitfähig?

Um in Anwendungen in Echtzeit auf Ereignisse reagieren zu können oder um Daten in Echtzeit bereitzustellen, reichen nichtechtzeitfähig – oder auch „konventionelle“ – Betriebssysteme oft nicht aus. Aufgrund der immer wieder gesteigerten Rechenleistung sind heutige Personalcomputer (PC) in der Lage komprimierte Musikinformationen in Echtzeit zu dekomprimieren und wiederzugeben. Dennoch sind jedem die dabei evtl. auftretenden ungewollten Unterbrechungspausen bekannt. Die Ursache für diese Tonaussetzer ist das Betriebssystem, das einem anderen Programm Ressouren zuteilt, die das Wiedergabeprogramm benötigt hätte. Man erkennt also leicht, daß konventionelle Betriebssystem weiche Echtzeitanforderungen unter Umständen bereits nicht erfüllen.

Wird der PC eingesetzt, um eine Lichtschranke zu überwachen, die die Hand eines Menschen vor den hohen Drücken einer industriellen Presse schützen soll, so wird besonders deutlich, daß das Betriebssystem auf gar keinen Fall dem falschen Prozess Rechnerresourcen zuteilen darf.

Um zu verstehen, warum konventionelle Betriebssysteme diese Echtzeitanforderungen nicht erfüllen, muß man sich ein wenig die Aufgaben eines Betriebssystems anschauen.

2.1 Aufgaben eines Betriebssystems

Zu den grundsätzlichen Aufgaben eines Betriebssystems gehört es für die zugrundeliegende Hardware eine Abstraktionsschicht bereitzustellen. Dies ist notwendig, damit den Anwendungsprogrammen eine einheitliche Schnittstelle bereitgestellt wird und die Komplexität der Hardware verborgen bleibt. Diese grundsätzliche Aufgabe läßt sich noch wie folgt aufgliedern, wobei ich nur Aspekte betrachte, die für Realzeitanwendungen von Bedeutung sind.

Resourcenverwaltung

In einem Rechner wird eine Vielzahl an Ressourcen zur Verfügung gestellt. Dazu gehören z.B. Datenspeicher, Grafikausgabe, Soundausgabe, Netzwerkzugriff und ähnliches. Aufgabe des Betriebssystem ist einerseits, wie bereits erwähnt, die Komplexität der Hardware zu verbergen. Ein gutes Beispiel dafür sind Ethernetkontroller. Der aktuellen Kernel 2.6.6 stellt 31 Treiberfamilien für 10 oder 100 MBit Ethernetkontroller bereit. Das Betriebssystem übernimmt die spezifische Ansteuerung des Kontrollers und stellt den Anwendungsprogrammen eine einheitliche Ethernetschnittstelle bereit. Anderseits muß das Betriebssystem den Zugriff auf die Hardware organisieren und koordinieren. Möchten zwei Anwendungsprogramm gleichzeitig Daten über das Netzwerk übertragen, so muß das Betriebssystem steuern, welches Programm wann auf das Netzwerk zugreifen darf.

Multitasking und Kontextwechsel

Ein besondere Aufgabe der Ressourcenverwaltung ist die Verwaltung der CPU bzw. der Rechenzeit und die Verteilung dieser an die Anwendungsprogramme.

Bei dieser Aufgabe unterstützen aktuelle Prozessoren das Betriebssystem, indem sie zwei spezielle Betriebsmodi zur Verfügung stellen. Unter Linux werden diese als Kernel Mode und User Mode bezeichnet. Im Kernel Mode läuft lediglich der Kernel von Linux. Der Kernel Mode erlaubt den Zugriff auf alle Register eines Prozessors und somit den Zugriff auf die Steuerung des Prozessors. Im User Mode laufen alle Anwendungsprogramme unter Linux. In diesem Modus ist der Zugriff auf die steuernden Register des Prozessors nicht erlaubt. Beide Betriebsmodi sind dazu da das Multitasking – also das scheinbar gleichzeitige Ausführen von Programmen bzw. Prozessen – in besonderem Maße zu unterstützen. Da lediglich der Kernel Zugriff auf die Steuerfunktionen hat ist er in der Lage sich selbst und Anwendungsprogramme untereinander vor Fehlern oder gar gezielten Angriffen zu schützen (Protection).

Multitasking und Protection bedeuten aber einen gewissen Aufwand in Form von Rechenzeit. Da alle steuernden Aufgaben nur vom Kernel durchgeführt werden, muß vom Programm in den Kernel gewechselt werden, also vom User Mode in den Kernel Mode. Dabei tritt jedesmal ein sogenannter Kontextwechsel – auch „system call trap“ genannt – auf. Das bedeutet, daß alle Register im Prozessor, die vom vorher laufenden Prozess benutzt wurden, vom Kernel gesichert werden müssen. Ansonsten würden beim Kontextwechsel die Inhalte einiger Variable verloren gehen. Sind die steuernden Aufgaben vom Kernel erledigt, wird ein Anwendungsprozess wieder auf den Prozessor gelassen. Dazu müssen alle Register für den kommenden Prozess geladen werden. Dieser Vorgang ist verständlich, wenn von einem Programm zum anderen gewechselt werden soll, er tritt aber wesentlich häufiger auf. Ein system call trap tritt im Prinzip jedesmal auf, sobald ein Programm auf eine Resource, also auf eine Funktion im Betriebssystem, zugreift.

Sogar die Hardware kann auf Funktionen im Kernel zugreifen, nämlich über Interrupts.

Interrupts

Interrupts werden von der Hardware an die CPU gesendet, um dem Betriebssystem zu signalisieren, daß es zur Steuerung der Hardware tätig werden muß. Meist wird einem Treiber signalisiert, daß die Ausgabe von Daten abgeschlossen ist oder Daten bereitstehen, die eingelesen werden sollen.

Dabei wird meist ein Prozess vom Prozessor verdrängt, so daß ein Kontextwechsel auftritt und schon einige Zeit vergeht, bis der Kernel auf den Interrupt reagiert.

Der Timerinterrupt wird dazu verwendet den Prozessen Rechenzeit zuzuteilen. Der Timerinterrupt tritt regelmäßig auf und bei jedem Auftritt kann der Kernel überprüfen, an welchen Prozess demnächst Rechenzeit vergeben wird.

Speicherverwaltung und Swapping

Eine weitere Aufgabe des Betriebssystems ist die Verwaltung des Arbeitsspeichers. Der im System vorhandene Arbeitsspeicher muß für die vielen Prozesse aufgeteilt werden.

Dafür wird der Speicher hierarchisch in mehreren Größenstufen in Seiten eingeteilt. Hierarchische Größenstufen sind notwendig, um den Speicherverbrauch für die Kontrolle dieser Seiten klein zu halten. Die Seiten werden dann an die Prozesse vergeben. Dabei steht jedem Prozeß im Prinzip der gesamte adressierbare Speicher von 4GB zur Verfügung. Aufgabe des Betriebssystem ist nun mit Hilfe der Memory Management Unit (MMU) im Prozessor nur die jeweils von einem Prozess benutzen Speicherseiten auf den real im System vorhandenen Arbeitsspeicher abzubilden.

Ist der real vorhandene Arbeitsspeicher zu klein um alle angeforderten Speicherseiten zu speichern, so kann der Kernel Speicherseiten, die lange nicht benutzt wurden auf ein anderes Medium – meist die Festplatte – auslagern. Das

hat den Vorteil, das der verfügbare Arbeitsspeicher innerhalb des Betriebssystem größer sein kann als der tatsächlich vorhandene. Der Nachteil ist, daß ein Prozess recht lange muß, bis er auf seinen Speicher zugreifen kann, weil das Betriebssystem diesen Teil ausgelagert hat.

Die Speicherverwaltung hat auch noch einen Nutzen für das Multitasking. Der Kernel wird sofort darüber informiert, wenn ein Prozess auf Speicher versucht zuzugreifen, den er vorher nicht angefordert hat. Dadurch wird vermieden, daß sich Programme gegenseitig negativ beeinflussen können.

Cold Caches

Der Zugriff von Prozessen auf den Arbeitsspeicher kann auch noch durch einen anderen Effekt als den der Auslagerung stark gebremst werden. Diesen Effekt bezeichnet man als „Cold Caches“.

Die Geschwindigkeit von Prozessoren entwickelt sich wie bisher schneller als die von Arbeitsspeichern. Deshalb ist der Einsatz von Caches notwendig, die schneller und kleiner sind als Arbeitsspeicher. Da Prozesse meist nur auf einen kleinen Teil im Arbeitsspeicher zugreifen, kann dieser Teil im Cache gehalten werden und so wesentlich schneller auf die Informationen zugegriffen werden. Wechselt das Betriebssystem nun zu häufig die Prozesse und greifen diese Prozess auf völlig verschiedene Arbeitsspeicherbereiche zu, so liegt der Bereich für einen Prozess oft nicht im Cache und es muß auf den langsameren Arbeitsspeicher zugegriffen werden.

2.2 Optimierungen konventioneller Betriebssysteme

Einem Betriebssystem hat, wie in Abschnitt 2.1 aufgeführt, viele Aufgabe zu erledigen. Dabei soll ein Betriebssystem aber auch einen möglichst weiten Einsatzbereich haben. Das hat zur Folge, daß Betriebssysteme auf den durchschnittlichen Fall hin optimiert werden. Dabei betrachtet man folgende Zielsetzungen:

Maximale Ressourcenauslastung Jede Ressource im Rechner soll bis an die durch die Hardware gesetzte Grenze ausgelastet werden. Beispielweise sollte eine Datei über ein 100 MBit Netzwerk mit 10 MByte/s übertragen werden oder von einer Festplatte, die eine Lesegeschwindigkeit von 10 MByte/s hat, sollte mit 10 MByte/s gelesen werden können, weil der Anwender darauf wartet, das die Übertragung abgeschlossen wird. Um dies zu erreichen, muß das Betriebssystem immer zum richtigen Zeitpunkt die dafür notwendigen Aktionen ausführen, also bespielsweise ohne Verzögerung auf Interrupts reagieren. Dazu muß ein laufender Prozeß natürlich unterbrochen werden.

Maximaler Durchsatz Darunter versteht man, daß so viel Programm wie möglich innerhalb eines gewissen Zeitraumes abgearbeitet werden soll. Denn aufwendige Berechnungen sollen so schnell wie möglich abgeschlossen sein.

Minimale Antwortzeit Die Antwortzeiten des Systems sollen kurz sein, denn der Anwender erwartet nach dem Bewegen der Maus oder nach einem Tastendruck sofort eine Reaktion auf dem Bildschirm.

Effizientes Scheduling Das Wechseln von Prozessen muß einerseits schnell gehen und andererseits muß die Zuteilung von Rechenzeit entsprechend den Erwartungen des Anwenders sein. Der Scheduler muß also einen genauen Überblick über bisher vergebene Rechenzeit schnell berechnen, um damit möglichst genau vorherzusagen, welcher Prozeß weiterhin viel Rechenzeit benötigen wird. Dazu gehört auch, daß Prozesse so häufig gewechselt werden müssen, daß der Schein der Nebenläufigkeit gewahrt bleibt.

Dabei fällt auf, daß die Optimierung hinsichtlich der obigen Kriterien bereits schon nicht möglich ist, weil sie sich bei der Realisierung widersprechen. Häufiges Wechseln von Prozessen zieht einen gewissen Overhead im Betriebssystem nach sich, wodurch der Durchsatz bereits nicht mehr maximal ist. Linux wurde ebenfalls anhand der obigen Kriterien optimiert und erfüllt so die Anforderungen eines Desktopbetriebssystems, auf dem meist nur ein Programm läuft, mit dem der Anwender interagiert, als auch die Anforderungen eines Serverbetriebssystems, auf dem viele Prozesse möglichst effizient die Resourcen zugeteilt bekommen.

2.3 Anforderungen für ein echtzeitfähiges Linux

Die bereits genannten Anforderungen an ein Echtzeitsystem – sofortige Abarbeitung von Daten und Bereitstellung von Ergebnissen zu einem bestimmten Zeitpunkt (Deadline) – lassen sich für Linux als Betriebssystem genauer aufschlüsseln, wobei die von Linux bereitgestellte Funktionalität nicht verloren gehen soll:

Optimierung auf den "Worst Case" Damit Linux in Echtzeit auf Ereignisse reagieren kann, muß die Optimierung auf den durchschnittlichen Fall wegfallen und durch eine Optimierung auf den Worst Case ersetzt werden. Hierbei handelt es sich nur um eine allgemeine Anforderung.

Deterministisches Verhalten Man muß versuchen das Scheduling deterministisch zu gestalten, um sicherzustellen, daß Aufgaben zu den Deadlines fertiggestellt sind. Aufgrund der Vielfalt der Konstellationen beim Scheduling wird ein vollständig deterministisches Verhalten nicht erreicht werden können. Das effiziente Scheduling muß durch ein vorhersagbares Scheduling ersetzt werden.

Kurze Latenzzeiten Kurze Latenzzeiten vereinfachen die Einhaltung der Deadlines. Latenzen treten durch Kontextwechsel, Kalte Caches und evtl. Swapping auf und sind unter Umständen gar nicht genau bestimmbar. Deshalb ist es um so wichtiger diese Latenzen kurz zu halten, um die Zeit zwischen geplanter und tatsächlicher Ausführung zu minimieren.

Echtzeitprozesse neben normalen Prozessen Die Unterstützung von Echtzeitprozessen neben normalen Prozessen ermöglicht, daß auf dem Echtzeitsystem sowohl zeitkritische Aufgaben als auch zeitlich unkritische Aufgaben erfüllt werden können. Zeitkritische Aufgaben können von den unkritischen Teilaufgaben befreit werden. Sie sind dadurch schneller ausführbar und das Gesamtsystem wird akkurater.

Kommunikation von Echtzeit- und Nichtechtzeitprozessen Direkt aus der Anforderung Echtzeitprozesse und normale Prozesse zu unterstützen folgt die Anforderung, daß diese auch kommunizieren können, damit die Teilaufgaben untereinander koordiniert werden können. Die Kommunikation beider Prozesstypen darf die Echtzeitprozesse natürlich nicht behindern.

Synchrone und Asynchrone Echtzeitprozesse Echtzeitprozesse müssen synchron und asynchron gestartet werden können. Synchrone Prozesse werden in regelmäßigen Zeitabständen gestartet, während asynchrone Prozesse völlig unregelmäßig, beispielsweise durch Interrupts gestartet werden.

Wenn man die Anforderungen optimal lösen möchte, entstehen bis heute nicht lösbare Probleme. Die Berechnung für die Verteilung der Rechenzeit auf die Echtzeitprozesse, so daß diese ihre Deadline einhalten, is NP-vollständig. Ein Scheduler mit NP-vollständigen Algorithmus würde zuviel Rechenzeit verbrauchen, so daß man sich mit anderen Algorithmen behelfen muß.

2.4 Scheduling unter Linux

Linux nutzt einen modularen Ansatz, um verschiedenen Prozessen verschiedene Scheduling-Verfahren zur Verfügung zu stellen. In einem Standard-Linux-Kernel gibt es drei Scheduling-Algorithmen:

SCHED_OTHER Der normal Time-Sharing-Scheduler. Dieser wird standardmäßig für Prozesse verwendet. Den Prozessen können Prioritäten von -20 bis 20 zugewiesen werden, wobei 0 die Ausgangspriorität ist. 20 ist die niedrigste Priorität und -20 die höchste. Zusätzlich werden diese statischen Prioritäten durch dynamische Werte angepaßt. Die dynamische Priorität des aktuell laufenden Prozess wird erhöht, damit dieser schwerer verdrängt werden kann, denn eine Verdrängung hätte einen Kontextwechsel zur Folge und der Durchsatz des Systems würde gemindert. Desweiteren wird dementsprechend wie ein Prozess seine Zeitscheibe ausnutzt die dynamische Priorität angepaßt, damit häufig blockierende Prozesse nicht benachteiligt werden.

SCHED_FIFO und SCHED_RR Diese beiden Schedulerstrategien sind für weiche Echtzeitprozesse vorgesehen. Es gibt jedoch keine Unterstützung für Deadlines oder ähnliches. Das gesamte Timing ist vom Prozess selbst zu managen. Innerhalb dieser beiden Schedulerstrategien gibt es auch Prioritäten. Der Prozess mit der höchsten Priorität wird dem Prozessor zugewiesen. Bei SCHED_FIFO wird von zwei Prozessen mit der gleicher Priorität derjenige auf den Prozessor gelassen, der zuerst gekommen ist. Dieser bleibt solange auf dem Prozessor bis er sich beendet. Bei SCHED_RR werden Prozesse mit der gleichen Priorität nach dem Round-Robin-Verfahren für jeweils eine Zeitscheibe auf den Prozessor gelassen.

Alle Prozesse werden in der gleichen Warteschlange eingereiht, so daß der Scheduler die gesamte Warteschlange durchsuchen muß, um einen Prozeß Rechenzeit zuteilen zu können. Somit hat der Scheduler von Linux eine Komplexität der Klasse O(n).

3 Wie kann man Realzeitbetriebssysteme implementieren?

Die Theorie für Realzeitsysteme hat vier verschiedene Ansätze hervorgebracht. Diese vier verschiedene Paradigmen, nach denen man Scheduling in Echtzeitbetriebssystemen realisieren kann, sehen wie folgt aus:

1. Das Scheduling wird anhand von statischen Einträgen gemacht. Diese werden bei der Implementierung des gesamten Realzeitsystems berechnet, wobei die Laufzeiten und die Deadlines aller Echtzeitprozesse berücksichtigt werden. Zur Laufzeit läßt sich an dem Scheduling nichts mehr verändern. Dies hat ein statisches System zur Folge, was für KURT nicht in Frage kommt.

2. Der zweite Ansatz verwendet Prioritäten und preemptives Scheduling. Zu diesem Ansatz zählen die Verfahren „Rate-Monotonic“ und „Earliest-Deadline-First“. Rate-Monotonic vergibt die Prioritäten proportional zur Laufzeit eines Prozesses, während Earliest-Deadline-First die Priorität einen Prozesses umso weiter erhöht, je näher er an seine Deadline kommt. Beide Verfahren können eine Einhaltung von Deadlines nicht garantieren.

3. Ein weitere Ansatz besteht darin ein konventionelles Scheduling zu nehmen und das System nur wenig auszulasten, damit jederzeit möglichst viel Rechenzeit zur Verfügung steht. Aufgrund dieser vielen freien Rechenzeit können Prozesse versuchen ihre Deadlines einzuhalten. Es wird ebenfalls keine Garantie für die Einhaltung der Deadlines gegeben.

4. Der letzte Ansatz setzt ebenfalls wie der erste feste Einträge für das Scheduling ein, jedoch werden diese Einträge zu dem Zeitpunkt berechnet, an dem der zugehörige Prozess im System angemeldet wird. Dabei kann überwacht werden, ob genügend Rechenzeit für einen neuen Echtzeitprozeß zur Verfügung steht und der Prozeß kann abgewiesen werden, falls dies nicht der Fall ist. Dieser Ansatz bietet die besten Möglichkeiten, um die Einhaltung von Deadlines zu gewährleisten.

3.1 Welche Projekte implementieren Echtzeitfähigkeit für Linux?

Es gibt zur Zeit drei konkurrierende Projekte, die Linux echtzeitfähig machen. Dazu zählt RTLinux von der Firma FSMLabs, RTAI (Real Time Application Interface) von DIAPM (Dipartimento di Ingegneria Aerospaziale - Politecnico di Milano) und KURT (Kansas University Real Time), entwickelt von dem Information and Telecommunication Technology Center (ITTC) an der University of Kansas.

KURT ist das am besten dokumentierte Projekt, so daß sich anhand dieses Projektes die notwendigen Änderungen am Linux-Kernel am besten aufzeigen lassen.

3.2 KURT

KURT wurde in 1998 auf dem Entwicklungskernel mit der Version 2.1 entwickelt und erfuhr in der Version 1.x noch einige Verbesserungen. Im Dezember 1999 wurde der Patch für die Kernel 2.2.5, 2.2.9 und 2.2.13 bereitgestellt, wobei KURT die Version 2.0 bekam. Im November 2001 wurde die Portierung auf den Kernel 2.4 fertiggestellt.

Die zugrundeliegende Architektur wurde seit der Version 1.0 nicht verändert, wobei diese in zwei Schritten entwickelt wurde. Sie wird im folgenden vorgestellt.

3.2.1 Verfeinerung der zeitlichen Auflösung mit UTIME

Die zeitliche Planung wird unter Linux mit Hilfe von Timern realisiert. Für jeden Timer wird eine Struktur angelegt, die den Zeitpunkt der Fälligkeit des Timers und den dazu gehörigen Funktionspointer enthält. Alle Timer werden in einer doppelt verketteten Liste gespeichert, wobei die Liste nach dem Zeitpunkt sortiert ist. Dies hat zur Folge, daß das Einfügen von Timer die Komplexität O(n) besitzt. Fällige Timer können jedoch in `O(1)` gefunden werden, da sich diese am Anfang der Liste befinden.

Die kleinste Zeiteinheit im Linuxkern beträgt auf x86-kompatiblen Prozessoren 10ms. Diese Zeiteinheit wird als „jiffy“ bezeichnet. Der Timerinterruptbaustein ist unter Linux so programmiert, daß er genau einmal pro jiffy einen Timerinterrupt auslöst. Somit bestimmt der jiffy die zeitliche Auflösung der Kernels.

Listing 1: Pseudo-Code für die Linux Timer Interrupt Service Routine

```
void timer_interrupt() {
        jiffies++;
        run_timer_list();
        update_process_times();
        ...
        schedule_process();
}
```

Listing 1 zeigt den Pseudo-Code für die Linux Timer Interrupt Service Routine (TISR). In dieser wird zuerst die Variable `jiffies` inkrementiert. Das Zählen der `jiffies` ergibt die Softwareuhr. Danach wird der Kopf der Timerliste nach fälligen Timern durchsucht und diese werden ausgeführt. Danach werden einige andere Daten aktualisiert und zuletzt wird ein Prozess ausgewählt, dem nach der TISR Rechenzeit zugewiesen wird. Man erkennt, daß der Timerinterrupt alle 10ms einerseits der Herzschlag (heartbeat) des Systems ist und daß Timer eine maximale Auflösung von 10ms haben. Dies reicht für Echtzeitsystem nicht aus.

Der erste, naive Ansatz wäre also die Auflösung zu erhöhen und somit die Periode des Timerinterrupts kürzer zu wählen. Sinnvoll für ein Echtzeitsystem wäre eine Auflösung von 1μs. Dies hat jedoch die negative Auswirkungen. Der Kernel muß 10000 mal öfter die TISR durchlaufen, was einen nicht mehr zu vertretenden Overhead produzieren würde bzw. die TISR würde immer wieder durchlaufen, wenn sie länger als 1μs dauert.

Es muß möglich sein, daß Timer genau auf die Mikrosekunde geplant werden können, wobei jedoch nicht zwingend jede Mikrosekunde ein Timer vorhanden ist. Also muß auch nicht zwingend jede Mikrosekunde ein Timerinterrupt auftreten. Dies wird von UTIME dadurch erreicht, daß der feste Timerinterrupt alle 10ms abgeschaltet wird. Stattdessen wird der Timerinterruptbaustein (TIB) im sogenannten „One Shot“-Modus betrieben. Es wird genau der Zeitpunkt anhand der Timerliste berechnet, wann der nächste Timerinterrupt auftreten muß und dieser wird dann im TIB programmiert.

Dies hat zur Folge, daß die Softwareuhr nicht mehr funktioniert und die Auflösung der Variablen `jiffies` nicht mehr ausreicht. Die Variable `jiffies` bekommt deshalb noch die Variable `jiffies_u` für die Mikrosekunden innerhalb eines jiffy und die Variable `jiffies_intr` für das Überschreiten einer Zeitscheibengrenze beiseite gestellt. Zur Aktualisierung der Softwareuhr wird als erstes die Methode `update_time()` innerhalb der neuen TISR (Listing 2) ausgeführt. Diese nutzt den Time Stamp Counter (TSC) der CPU, welcher einmal pro Takt inkrementiert wird, um die Variable `jiffies_u` zu aktualisieren. Überschreitet `jiffies_u` die Länge einer Zeitscheibe, wird `jiffies` inkrementiert und `jiffies_intr` gesetzt.

Listing 2: Pseudo-Code für die UTIME Linux Timer Interrupt Service Routine

```
void timer_interrupt() {
        update_time();
        t = time_to_next_event();
        program_timer_chip(t);
        run_timer_list();
        if (jiffies_intr) {
                update_process_times();
                ...
                schedule_process();
                jiffies_intr = 0;
        }
}
```

In der ersten Implementierung wurde die Softwareuhr anhand der Differenz des TSC-Wertes zwischen zwei Timerinterrupts aktualisiert. Dabei stellte sich heraus, daß die Softwareuhr noch recht ungenau läuft. Diese Implementierung wurde geändert und es wird nun beim Booten ein Referenzwert des TSC und der Uhr gespeichert. Die Softwareuhr wird anhand der Differenz zwischen dem TSC-Wert zu einem Interrupt und dem Referenzwert aktualisiert. Dadurch wird die Softwareuhr wesentlich genauer.

Die Funktione `time_to_next_event()` berechnet den Zeitpunkt, an dem der nächste Timerinterrupt auftreten muß und mit `program_timer_chip()` wird dieser Zeitpunkt in den TIB programmiert. Danach wird wie gehabt die Timerliste nach abgelaufenen Timern durchsucht und diese werden ausgeführt. Anschließend wird nur bei Überschreiten einer Zeitscheibengrenze die Methoden für den Heartbeat des Systems ausgeführt. Damit der Heartbeat nicht verloren geht, wenn keine Timer vorhanden sind, wird mindestens einmal pro Jiffy ein Pseudo-Timer programmiert. Würde dies nicht gemacht, so ist z.B. vom Multitasking nicht mehr viel zu spüren, weil die Prozesse zu langsam gewechselt werden.

Die erste Implementierung sah eine Neuprogrammierung des TIB bei jedem Lauf der TISR vor. Die wurde dahingehend verbessert, daß der TIB nur dann neu programmiert wird, wenn der nächste Timer nicht kurz auf den aktuellen folgt, sondern es wird einfach der entsprechende Zeitraum mit Hilfe des TSC gewartet.

3.2.2 Leistungsfähigkeit von UTIME

Die Leistungsfähigkeit von UTIME zeigen die drei folgenden Messungen. Die erste Messung dokumentiert die Laufzeit[1] der neuen TISR von UTIME.

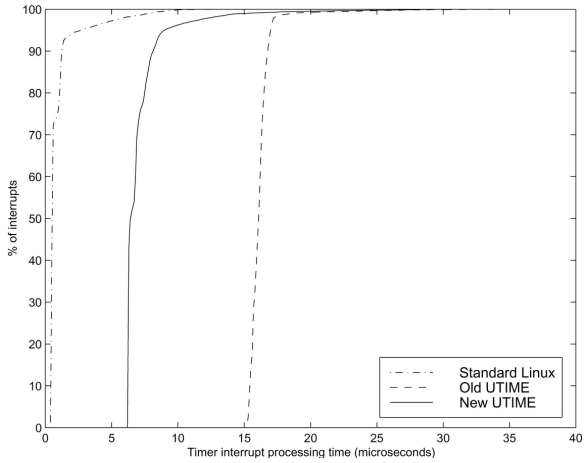


Abbildung 1: Laufzeitvergleich der Timerinterrupt Service Routinen

Die Abbildung 1 zeigt den prozentualen Anteil der Durchläufe der TISR in Abhängigkeit von der Durchlaufzeit. Die Linie stellt dar, welcher Anteil innerhalb einer bestimmten Laufzeit liegt. Die meisten Timer Interrupts konnten mit der originalen TISR innerhalb 1μs abgearbeitet werden. Alle waren jedoch unterhalb von 5μs. UTIME erhöht leider die Laufzeit durch die TISR auf ca. 15μs. Ursache dafür sind die Berechnungen für die Softwareuhr und für den nächsten Timerinterrupt als auch die Programmierung des TIB. Ein weitere Messung ergab, daß die Programmierung des TIB alleine 4μs benötigt. Somit ist auch klar, warum bei der Neuimplementierung der TISR nur noch ca. 7μs benötigt.

Abbild 2 zeigt die Abweichung der Softwareuhr unter Standard Linux, unter UTIME und unter dem verbesserten UTIME. Man kann deutlich erkennen, daß der TSC wesentlich genauer arbeitet, als der TIB und daß bei der Verbesserung der Architektur nocheinmal die Genauigkeit stark gesteigert werden konnte.

Um die Verbesserung der Auflösung von Timern zu messen, wird ein Prozess verwendet, der alle 10ms eine Schleife durchlaufen soll. Die Genauigkeit der Einhaltung von 10ms gibt einen Aufschluß darüber, inwieweit sich die Auflösung der Timer im Kernel verbessert hat. Der Prozess läuft unter der Schedulerstrategie SCHED_FIFO.

Die Abbildung 3 zeigt unter (a) die Genauigkeit der Schleife mit den Veränderungen von UTIME im Kernel und unter (b) die Genauigkeit mit einem Standardkernel. Da in einem Standardkernel der Timerinterrupt selbst eine Auflösung von 10ms hat, wird der Timer für die Messschleife nur alle zwei Timerinterrupts ausgeführt. Dadurch wird die Schleife nur alle 20ms durchlaufen. Im Teilbild (a) ist deutlich zu erkennen, daß die Einhaltung von 10ms innerhalb der Schleife sehr gut eingehalten werden. Die Verbesserungen der zweiten Implementierung lassen die Kurve noch etwas steiler werden.

[1] Alle Messungen wurden auf Systemen mit einem 133MHz oder einem 200MHz Pentium Prozessor durchgeführt.

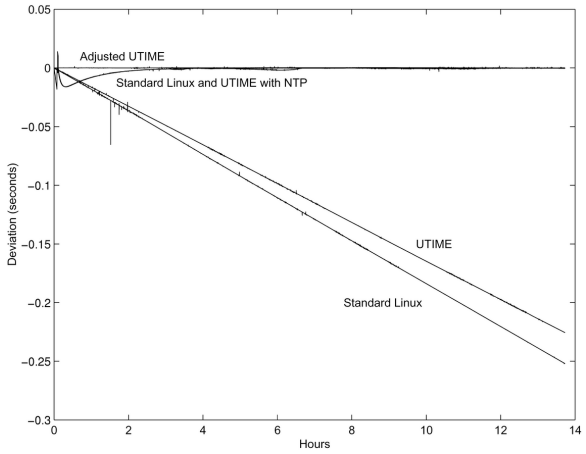


Abbildung 2: Abweichung der Softwareuhr

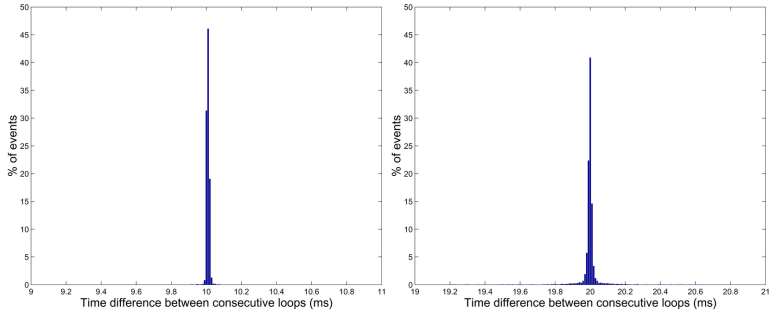
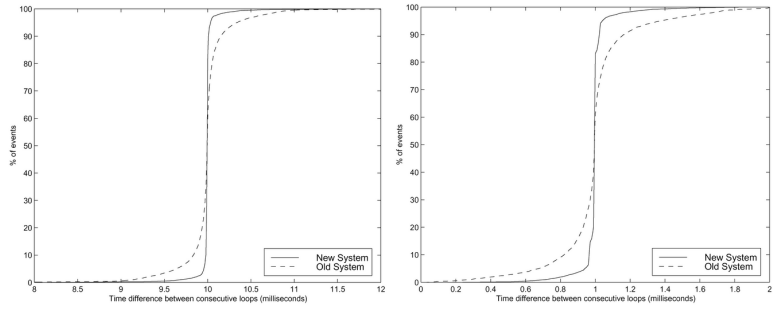


Abbildung 3: Einhaltung einer Schleifenzeit von 10ms



(a) 10 millisecond period (b) 1 millisecond period

Abbildung 4: 10ms-Schleife nach der Verbesserung

Die letze Messung dokumentiert die Zeit, die zwischen dem Timerinterrupt und dem Eintritt in die zum Timer gehörige Funktion vergeht. Der dazu eingesetzte Prozess verwendet die Schedulerstrategie SCHED_FIFO.

Die Abbildung 5 zeigt innerhalb welcher Zeit die Routine nach dem Timerinterrupt betreten wird. Man kann erkennen, daß zu den 15μs für die TISR noch ca. 10μs dazukommen bis im schnellstmöglichen Fall die Funktion angesprungen wird. Maximal vergehen 50μs zwischen Interrupt und Eintritt in die Funktion. Die zusätzlich Zeit zur Laufzeit der TISR wird durch den Kontextwechsel und die Umschaltung von Kernel Mode in den User Mode verursacht. Desweiteren spielen hier die CPU-Caches eine Rolle. Muß die CPU Daten oder Befehle aus dem Arbeitsspeicher anstatt

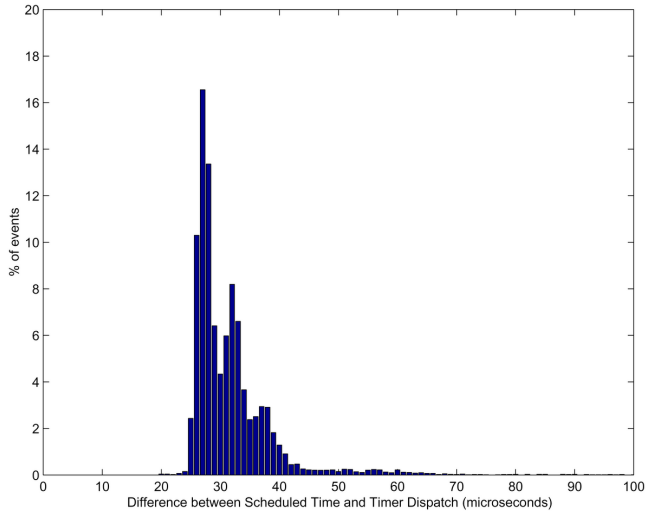


Abbildung 5: Zeitmessung vom Interrupt zur eingeplanten Routine

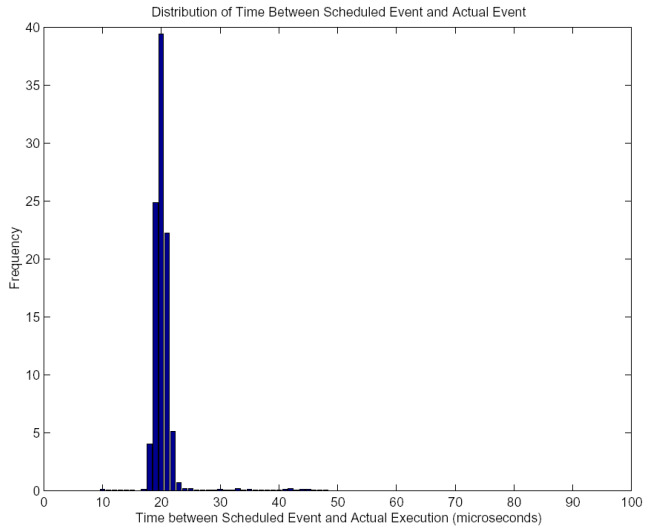

Distribution of Time Between Scheduled Event and Actual Event
Frequency
Time between Scheduled Event and Actual Execution (microseconds)

Abbildung 6: Zeitmessung vom Interrupt zur eingeplanten Routine (Verbesserung)

aus dem Cache lesen, so wirkt sich dies auf die Laufzeit aus.

Die maximale Zeit von 50μs muß bei der Planung von Deadlines für Echtzeitprozesse berücksichtigt werden. Die Abbildung 6 zeigt die Verzögerung vom Interrupt bis zum Eintritt in die zum Timer zugehörige Funktion nach der Verbesserung der Architektur. Durch die Beschleunigung der TISR wird die Verzögerung deutlich verkleinert.

3.2.3 Realzeitscheduling mit KURT

KURT ist in zwei Teile aufgeteilt, und zwar in den KURT Core und die RTMods.

KURT Core

Der Kern von KURT ist für das Scheduling der Echtzeitevents (RTEvents) zuständig. Er startet das entsprechende RTMod zur entsprechenden Zeit. Um dies zu erreichen, wird das Scheduling mit Hilfe eines eindeutigen Plans

durchgeführt. Es handelt sich also um einen sogenannten „explicit plan scheduler“. Dieses Paradigma wurde gewählt, weil er zur Zeit das einzige ist, mit dem ein berechenbares Verhalten für Echtzeitprozesse erreicht werden kann.

RTEvents

Die RTEvents für jeden Echtzeitprozess (RTProzeß) müssen im Voraus definiert werden. Es gibt zwei Möglichkeiten dies zu tun:

1. Handelt es sich um einen Prozeß der nur sporadisch zu bestimmten Zeitpunkten gestartet werden muß, so werden diese Zeitpunkte beim Starten des Prozesses definiert. Diese definierten Zeitpunkte müssen nicht zwingend im Arbeitsspeicher (RT_FROM_MEM) abgelegt werden; es besteht auch die Möglichkeit diese Daten als Datei auf einem Datenträger (RT_FROM_DISK) abzulegen. Die Zeitpunkte werden dann vom Kern rechtzeitig geladen. Sind die RTEvents im Arbeitsspeicher abgelegt, besteht die Möglichkeit diese RTEvents zu wiederholen.
2. Für periodische Prozesse gibt es die Möglichkeit, daß zum Start des Prozesses nur die Dauer der Periode und die Anzahl der Wiederholungen spezifiziert wird. Der Kern plant dann für diesen periodischen Prozeß die entsprechenden RTEvents ein.

Für beide Möglichkeiten prüft der KURT Core, ob genügend Rechenzeit für die Echtzeitprozesse zur Verfügung steht. Bei periodischen Prozessen kann der Zeitpunkt für den ersten RTEvent definiert werden; wird dies nicht getan, bestimmt der Scheduler den Zeitpunkt für den ersten RTEvent, so daß der Prozeß bestmöglich in den Plan paßt. Zusätzlich zum Zeitpunkt für den RTEvent, muß die Dauer für die Abarbeitung eines RTEvents definiert werden, so daß der Scheduler in der Lage ist ein Überladen der CPU zu verhindern. Passen die Zeitpunkte für RTEvents oder die Abarbeitungszeit für einen neuen RTProzeß nicht mehr in den Plan, so weißt KURT diesen neuen Prozeß zurück.

Die Weiterentwicklung der Architektur ermöglichte zusätzlich zu den zeitgesteuerten RTProzessen eventgesteuerte. Eventgesteuert ist ein Prozeß dann, wenn er vom Betriebssystem so lange blockiert wird, bis das betreffende Ereignis eintrifft. KURT nimmt eventgesteuerte Prozesse ohne Prüfung des Scheduling-Plans an. Erst in dem Moment, in dem das Ereignis eintrifft, prüft der Scheduler, ob im Plan ausreichend Rechenzeit für den eventgesteuerten RT-Prozeß frei ist und führt ihn gegebenenfalls aus. Ist nicht ausreichend Rechenzeit vorhanden, so bleibt der RTProzeß bis zu einer Lücke oder bis zu einem Timeout blockiert.

Wurde ein neuer RTProzess erfolgreich von KURT angenommen, so legt der Kern für die RTEvents Timer im Linuxkernel an, wobei dieser Timer 50μs vor dem RTEvent gesetzt wird. Sollte der Eintritt in den RTProzeß vor schneller als in 50μs erreicht werden, so wird der TSC der CPU verwendet, um den Eintritt bis zum eigentlichen RTEvent zu verzögern. Dieses wird gemacht, weil die Zeiten vom Timerinterrupt bis zum Eintritt in den RTProzeß wie oben beschrieben unterschiedlich sein können. Diese Schwankungen müssen weitesgehend unterdrückt werden.

Betriebsmodi

Primär werden zwei Betriebsmodi in KURT eingesetzt. Es gibt den normalen Modus, in dem der Kernel Realzeitprozesse nicht berücksichtigt und wie gewohnt den Time-Sharing-Scheduler für konventionelle Prozesse einsetzt, und es gibt den Realzeitmodus, in dem über den Plan Realzeitprozesse und über den Time-Sharing-Scheduler konventionelle Prozesse ausgeführt werden.

Bei der Implementierung wurde festgestellt, daß einige Subsysteme im Kernel für recht lange Zeiträume Interrupts sperren. Die Abbildung 7 zeigt die Messungen diesbezüglich. Es fällt besonders das Disk-Subsystem mit Sperren der Interrupts für bis zu 250μs auf. Folglich ist ein Prozeß in der Lage beim Zugriff auf Datenträger den Timerinterrupt für maximal 250μs zu blockieren und somit evtl. die Ausführung von Echtzeitprozessen zu stören. Damit der Entwickler dies unterbinden kann, kann das Scheduling eingestellt werden. Wird für den Echtzeitscheduler `SCHED_KURT_PROCS` angegeben, so wird die Ausführung von normalen Prozessen deaktiviert, während bei `SCHED_ALL_PROCS` normale Prozesse neben Echtzeitprozessen laufen dürfen.

Es stellte sich jedoch heraus, daß diese Unterteilung zu grob war und die Verbesserung der Architektur bringt die Möglichkeit mit sich einzelne Subsysteme des Kernels zeitweise zu deaktivieren. Damit kann der Entwickler genauer die Abläufe im Realzeitsystem steuern.

RTMods und Process RTMod

KURT bietet zwei Möglichkeiten um Realzeitprozesse zu implementieren. Wird der Realzeitprozess als RTMod implementiert, so befindet sich der RTProzeß innerhalb eines Kernelmoduls. Dieses Kernelmodul kann zur Laufzeit zum

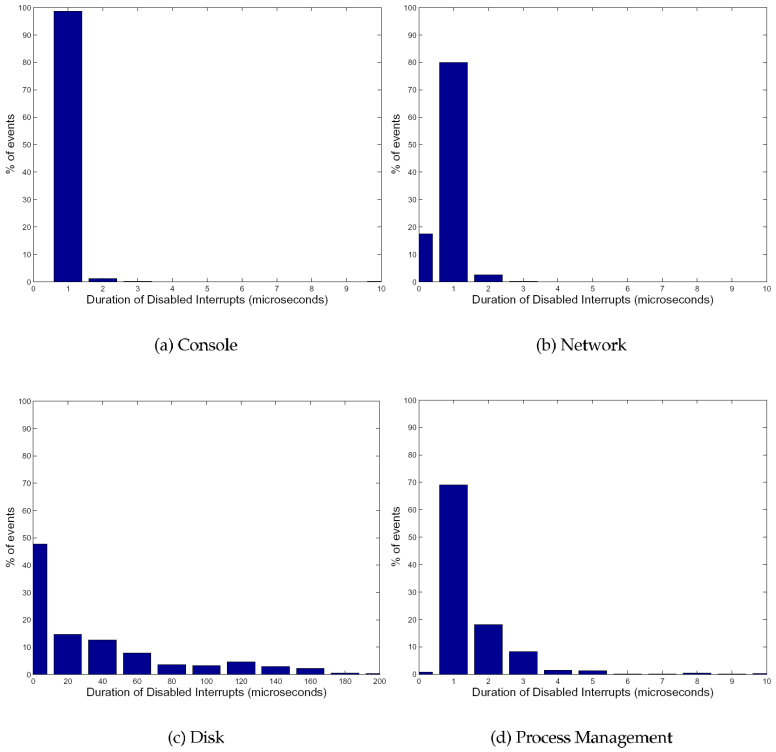


Abbildung 7: Dauer der Sperrung von Interrupts

Kernel hinzugeladen werden. Es ergeben sich dadurch eine Reihe von Vorteilen, die sich positiv auf das zeitliche Verhalten der Prozesse auswirken:

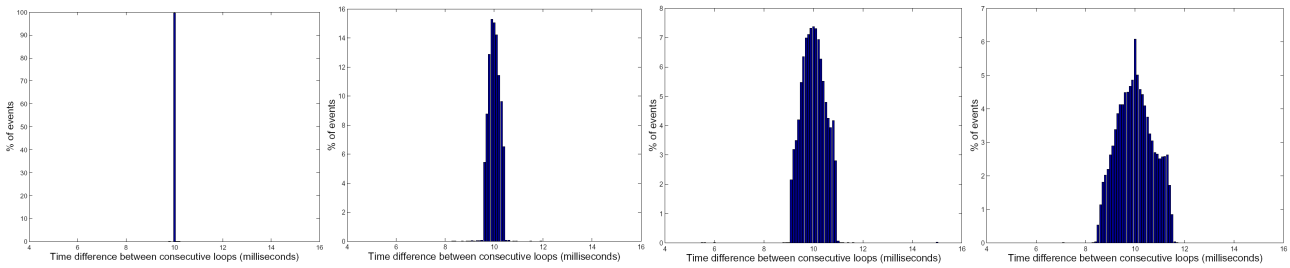
- RTMods werden als Kernelmodule im gleichen Adressraum wie der Linuxkern ausgeführt. Dadurch entfällt der Kontextwechsel zwischen Kernelfunktionen und Funktionen des RTMods. Die Zeiten vom Timerinterrupt bis zum Eintritt in das RTMod wird kürzer.
- Das RTMod wird im Kernel Mode ausgeführt, so daß auf Befehle der CPU zurückgegriffen werden kann, die nur im Kernel Mode zur Verfügung stehen.
- Ein RTMod kann auf alle Services der Linuxkernels zugreifen. Dieser Zugriff ist schneller, weil der Kontextwechsel vermieden wird und es kann auf Funktionen zugegriffen werden, die einem normalen User-Mode-Prozeß nicht zur Verfügung stehen.

RTMods haben aber auch bei der Entwicklung einen entscheidenden Nachteil. Hat das RTMod einen Programmierfehler, so stürzt bei Auftreten des Fehlers nicht nur der Realzeitprozeß ab, sondern das gesamte Linuxsystem.

Deshalb gibt es zur Vereinfachung der Entwicklung von Realzeitprozessen ein spezielles RTMod, genannt „Process RTMod“. Es führt selbst keinen Realzeitprozeß aus, sondern verteilt Rechenzeit für Realzeit an gekennzeichnete User-Mode-RTProzesse. Dadurch ist der Entwickler in der Lage Realzeitprozesse bei der Entwicklung im User Mode laufen zu lassen, wobei er auf alle Realzeitfunktion von KURT Zugriff hat. Stürzt der User-Mode-RTProzeß ab, so kann das restliche Linuxsystem weiterlaufen. Nachteilig bei User-Mode-RTProzessen ist jedoch eine wesentlich höhere Latenz als bei RTMods, die durch Kontextwechsel und den Overhead von System Calls verursacht wird.

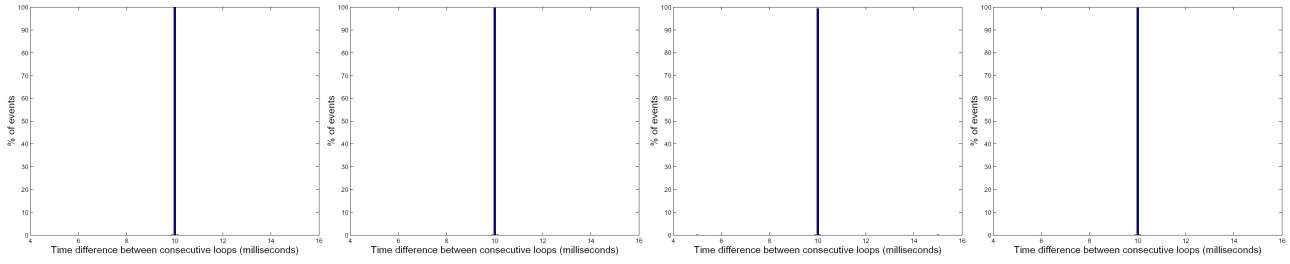
3.2.4 Performance von KURT

Die Performance von KURT wird ebenso wie der Performance von UTIME mit einer leeren Schleife gemessen, die alle 10ms durchlaufen werden soll. Dabei läßt man 1, 10, 20 oder 30 dieser Prozesse gleichzeitig laufen.



(a) 1 Process (b) 10 Process (c) 20 Process (d) 30 Process

Abbildung 8: Effektivität des Scheduler SCHED_FIFO

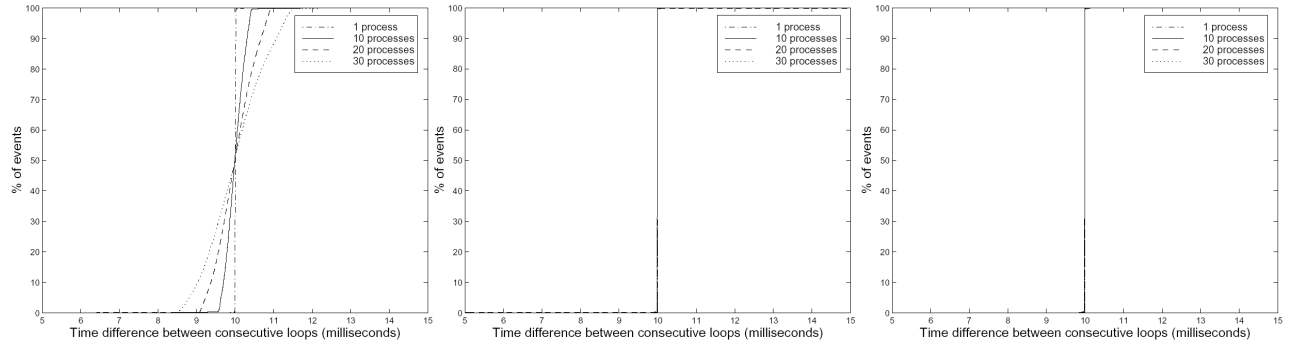


(a) 1 Process (b) 10 Process (c) 20 Process (d) 30 Process

Abbildung 9: Effektivität des Scheduler SCHED_KURT

Abbild 8 zeigt, wie genau die 10ms vom FIFO-Scheduler des Linuxkerns eingehalten werden. Es ist deutlich zu erkennen, daß mit zunehmender Prozeßanzahl die zeitliche Genauigkeit immer schlechter wird.

Abbildung 9 zeigt die Ergebnisse des gleichen Tests, jedoch wird diesmal der Scheduler von KURT für die Realzeitprozesse eingesetzt. Der Scheduler ist in der Lage sogar 30 Prozesse alle 10ms laufen zu lassen. Bei der Registrierung der Prozesse wurde eine Laufzeit von 300μs angegeben, so daß das System nahezu vollausgelastet ist. KURT verweigert gemäß seinem Plan das Starten von weiteren Prozessen, um das System nicht zu überlasten. Der FIFO-Scheduler läßt auch bei Vollauslastung weitere Prozessen zu.



(a) FIFO (b) KURT (SCHED_KURT_PROCS Mode) (c) KURT (SCHED_ALL_PROCS Mode)

Abbildung 10: Effektivität der Scheduler im Vergleich

Abbildung 10 zeigt nochmal die Genauigkeit des FIFO-Scheduler im Vergleich zum KURT-Scheduler, wobei beim KURT-Scheduler unter (b) nur Echtzeitprozesse zugelassen werden, während unter (c) im Hintergrund zusätzlich

normale Prozesse von Linux ausgeführt werden. Normale Prozesse beeinflussen den KURT-Scheduler bei seiner Arbeit nicht, solange nicht über längere Zeiträume die Interrupts deaktiviert werden.

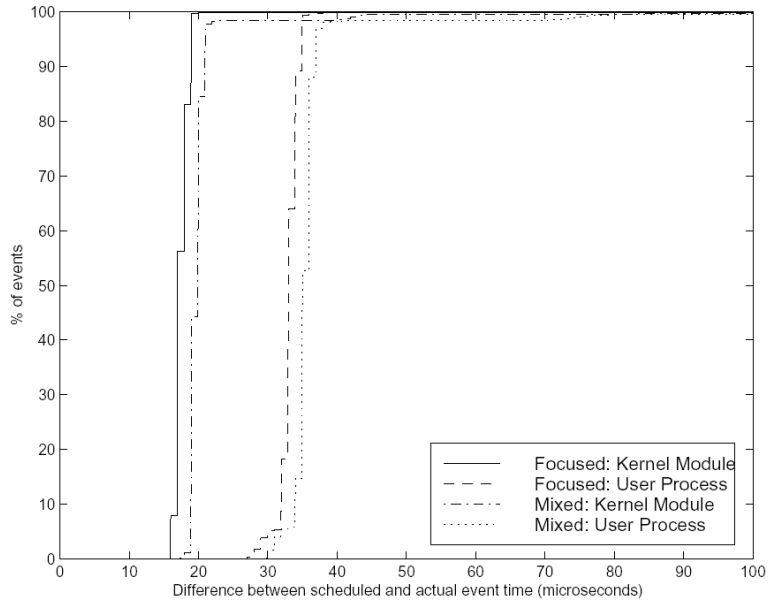


Abbildung 11: SCHED_ALL_PROCS und SCHED_KURT_PROCS im Vergleich

Den Vergleich von RTMods und User-Mode-RTProzessen zeigt Abbildung 11. Zusätzlich wird `SCHED_KURT_PROCS` (Focused) und `SCHED_ALL_PROCS` (Mixed) des KURT-Scheduler unterschieden. Gut zu erkennen ist, daß User-Mode-RTProzesse durchschnittliche 14μs später ausgeführt werden als RTMods, was durch den Kontextwechsel zu erklären ist.

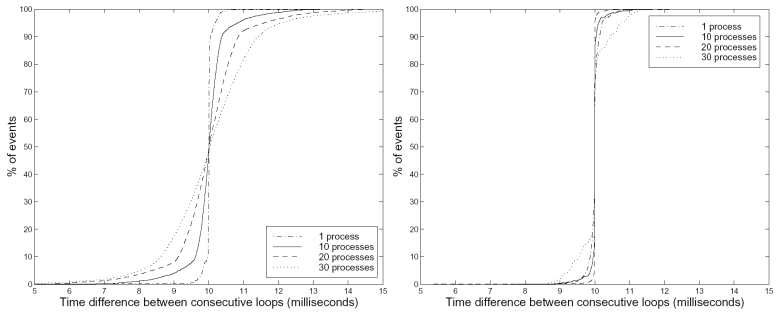


Abbildung 12: Zeiten mit Hintergrund-I/O-Aktivität

Das Problem der gesperrten Interrupts verdeutlich Abbildung 12. Die linke Grafik stellt die Genauigkeit des KURT-Schedulers dar, wobei von einem normalen Prozeß auf die Festplatte zugegriffen wurde. Rechts der gleiche Test ohne Festplatteaktivität im Hintergrund. Die Sperrung von Interrupts durch das Disk-Subsystem hat zur Folge, daß die RTEvents deutlich schlechter eingehalten werden können.

4 Ausblick

KURT erzielt bereits sehr gute Ergebnisse, um Prozesse in Realzeit auszuführen. Es können RTEvents mit einer Genauigkeit von 1μs definiert werden und sogar unter hoher Last werden Deadlines gut eingehalten aufgrund des

eindeutigen Plans des Schedulers. Dennoch bleibt ein wichtiges Problem. Die Sperrung von Interrupts stört deutlich die Einhaltung von RTEvents. In der Release Note des aktuellsten KURT für den Kernel 2.4.18 wird der „Preemption-Patch“ und der „Lock-Breaking-Patch“ vorausgesetzt. Der Lock-Breaking-Patch unterbindet Locks, die innerhalb vom Kernel lange gehalten werden, während der Preemption-Patch Interrupts in vielen Kernelfunktionen erlaubt, die vorher Interruptsperren hatten. Leider gibt es keine Dokumentation darüber, inwieweit die Einbindung dieser Patches Verbesserungen im System bewirken. Genausowenig ist die symmetrische Multiprozessorunterstützung dokumentiert. Der Kernel 2.6 beeinhaltet bereits diese Patches und noch einige Verbesserungen, die Latenzen im System vermindern. Eine Portierung von KURT auf den Kernel 2.6 könnte die Exaktheit des System weiter verbessern.

Literatur

[1] Nordmann, Michael (21.11.2001). Echtzeit für Linux. Ausarbeitung zum Seminar Linux und Apache, Fachhochschule Wedel. http://www.fh-wedel.de/~si/seminare/ws01/Ausarbeitung/6.linuxrt/LinuxRT0.htm

[2] Information and Telecommunication Technology Center (6.10.2003). KURT-Linux Homepage, University of Kansas. http://www.ittc.ukans.edu/kurt

[3] Information and Telecommunication Technology Center (24.3.2000). Old KURT-Linux Homepage, University of Kansas. http://www.ittc.ukans.edu/kurt/old-index.html

[4] Srinivasan, Balaji (1998). A Firm Real-Time System Implementation using Commercial Off-The-Shelf Hardware and Free Software, University of Kansas.

[5] Hill, Robert (1998). Improving Linux Real-Time Support: Scheduling, I/O Subsystem, and Network Quality of Service Integration, University of Kansas.

Real Time Java: Ein Überblick

Seminararbeit
im Fach Informatik
im Rahmen des Seminars "**Sicherheitskritische Systeme**"
an der

Universität Siegen, Fachgruppe für Praktische Informatik

eingereicht bei
Dr. Jörg Niere

vorgelegt von
Frank Christoph Köther

Sommersemester 2004

Real Time Java – Ein Überblick

Frank Ch. Köther

Wenn man sicherheitskritische Systeme betrachtet, kommt man an einer Programmiersprache, mit der man diese Systeme, bzw. das Verhalten dieser Systeme festlegen kann, nicht vorbei. In diesem Seminar wird ein Überblick über die Echtzeit-Programmiersprache Real time Java gegeben.

1 Einführung

Bei sicherheitskritischen Systemen wird verlangt, dass diese in einer festgelegten Zeitspanne und mit einer vorher genau definierten Aktion auf ein bestimmtes Ereignis reagieren. Es darf kein mögliches Ereignis unbehandelt bleiben, da es in diesen Systemen keine undefinierten Zustände geben darf. Die Aktionen dürfen wiederum auch nicht zu unbestimmten Zuständen führen. Es ist also extrem wichtig, dass man immer ohne Einschränkungen genau vorhersagen können muss, wie und wie schnell das System reagiert.

In der heutigen Zeit werden diese Anforderungen an Systeme nicht mehr, wie früher direkt in spezieller Hardware und schnellem Maschinencode umgesetzt, sondern zunehmend in Embedded Systemen. Diese bestehen meist aus allgemeiner, nicht zu hoch spezialisierter, Hardware, einem Betriebssystem und darauf laufenden Applikationen. Auf das Betriebssystem und dessen Anforderungen geht das Seminar „Sicherheitskritische Systeme: Real time Linux – Ein Überblick“ von Marcus Klein[1] genauer ein. Hier werde ich mich mit Real time Java (RTJ) beschäftigen. Mit dieser Echtzeit-Programmiersprache können Applikationen für diese Echtzeit-Systeme erstellt werden.

2 Warum wird Java als Basis verwendet?

Es gibt bereits Programmiersprachen, die für die Erstellung von Echtzeit-Anwendungen verwendet werden. Da existiert zum Beispiel Ada, dass bis jetzt noch häufig von den deutschen und amerikanischen Verteidigungsministerien für die Realisierung derartiger Softwareanforderungen verwendet wird. Aber Java bietet viele entscheidende Vorteile gegenüber anderen den anderen Programmiersprachen:

- Java wird von einer sehr breiten Anwendergemeinde verwendet.
- Java ist auf sehr vielen verschiedenen Plattformen, wie PDA, Handy, PC, usw. verfügbar und kann dort mit geringen, bis gar keinen Codeanpassungen sofort ausgeführt werden. Es gibt aber auch Einschränkungen, weil gerade im Bereich der Embedded Systeme sehr begrenzte Ressourcen für die Java Virtual Machine (JVM) zur Verfügung stehen. Dort gibt es zum Beispiel nur sehr leistungsschwache Benutzerschnittstellen (User Interfaces: UIs) im Gegensatz zur PC Welt. Ebenso sind Speicher und Rechenleistung meistens sehr knapp bemessen, so dass das Java nun doch mit anderen Bibliotheken, speziell für diese Umgebungen ausgestattet werden muss. Somit ist also ein Java-Programm nur eingeschränkt auf allen Plattformen nutzbar.

Die folgenden Punke gehören grob zu den Entwicklungsrichtlinen zu Java und beruhen auf dem Bestreben, nicht die gleichen oder ähnliche Fehler zu machen, wie sie bei der Erstellung anderer Programmiersprachen passiert sind:

- Die Problemlösung, für die der Code erstellt wird, wird auf einer hohen Abstraktionsebene beschrieben, weil Java eine Hochsprache ist. Die Sprache Java ist also Prozessorunabhängig, leichter lesbar für Menschen, es gibt Kontrollstrukturen, komplexere Datentypen, Syntax- und Typüberprüfung.
- Java ist leichter in vollem Umfang zu erlernen, wie C++, weil man bei der Spezifikation von Java darauf geachtet hat, die Fehler, die bei der Entwicklung von C, C++ und anderen Programmiersprachen gemacht worden sind zu vermeiden. Und man legte sehr viel Wert darauf, viele Sachen zu vereinfachen.

- Java ist relativ „sicher“, da es auf einer Virtual Machine (VM) läuft, die nur eingeschränkten Zugriff auf die Hardware und die Software zur Laufzeit hat. Der Code läuft quasi in einer „Sandbox“. Dieser Zugriff muss bei Real Time Java aber etwas gelockert sein gegenüber dem normalen Java, dazu aber mehr im weiteren Verlauf dieses Seminars.
- Java unterstützt das dynamische Laden neuer Klassen, so das zur Kompilezeit nicht alle Klassen in der endgültigen Form vorliegen müssen, sondern auch noch später zur Laufzeit nachgereicht bzw. aktualisiert werden können.
- Java unterstützt Objekt- und Thread-Erstellung zur Laufzeit. Man kann also diese Vorgänge direkt mit Java zur Laufzeit beeinflussen und von lokalen Situationen abhängig machen.
- Java wurde auch entworfen, um Code-Komponenten zu integrieren und wieder zu verwenden. Damit sich Funktionsbibliotheken anlegen und verbreiten lassen. Dies ist generell bei heutigen Programmiersprachen wichtig, damit dadurch Arbeitszeit gespart werden kann, weil nicht ständig „das Rad neu erfunden wird“. Zudem sind die Komponenten meist auch schon getestet.
- Java unterstützt verteilte Systeme. Die meisten modernen Systeme bestehen aus einem Netzwerk aktiver Komponenten, die zusammen für die Lösung von Aufgaben verwendet werden. Um eine sinnvolle Verteilung der Einzelaufgaben zu machen, muss man dieses Netzwerk als Ganzes sehen. Und damit ist es auch von großem Vorteil, dieses Netwerk bis zu einem gewissen Grad als Ganzes programmieren zu können.
- Java gewährleistet eine wohl definierte Ausführungssemantik, da jeder Befehl in allen Einzelheiten dokumentiert ist (JavaDoc). Diese Dokumentation enthält alle möglichen Parameter und deren Beschreibung, mögliche Ausnahmen und deren Quellen und vieles mehr. Alle diese Beschreibungen haben dasselbe Format. Es gibt keine versteckten Parameter und es wird genau beschrieben, was der jeweilige Befehl bewirkt.

3 Merkmale von Real time Java

Die Programmiersprache Java ist nicht als Echtzeit-Programmiersprache konzipiert worden, deshalb wurde ein Spracherweiterung RTJ für Java entwickelt, die in Verbindung mit einer neuen eigenen Virtual Machine(RTJVM), die Echtzeit-Anforderungen erfüllt.

Man hat versucht, sich bei der Entwicklung der Erweiterung an folgende Leitlinien zu halten:

- Die Erweiterung muss Abwärtskompatibilität gewährleisten, denn sonst würde man den alten Code gar nicht wieder verwenden können und man hätte auch eine neue Programmiersprache entwickeln können.
- Einmal Code schreiben und überall ausführen (WORA-Prinzip)
- Vorhersagbare Ausführung des Code
- Sofortige Verfügbarkeit und weitere Verbesserungen später. RTJ ist noch in der Entwicklungsphase, aber dennoch können schon die ersten Programme geschrieben werden, weil schon einige Grundprinzipien, die in der Spezifikation als „*Minimum Implementations of the RTSJ*“ genannt werden, umgesetzt worden sind.
- Keine syntaktischen Erweiterungen
- Erlaubt, wie Java selbst, verschiedene Variationen von Implementations-Entscheidungen

Das „ur“-Java musste in folgenden Bereichen überarbeitet werden, um den Echtzeitkriterien zu genügen:

3.1 Scheduling

Für die Echtzeitprogrammierung ist es extrem wichtig, dass Code in einem bestimmten zeitlichen Rahmen oder zu einer vorhersagbaren Zeit ausgeführt wird. Deshalb erben diese Klassen mit diesem Code in RTJ von dem Interface *Schedulable*. Damit wird gewährleistet, dass ein Scheduler mit diesen Threads zur Laufzeit umgehen kann. Aber das reicht noch nicht aus, dass diese Threads auch in der gewünschten Zeit und Reihenfolge abgearbeitet werden

.

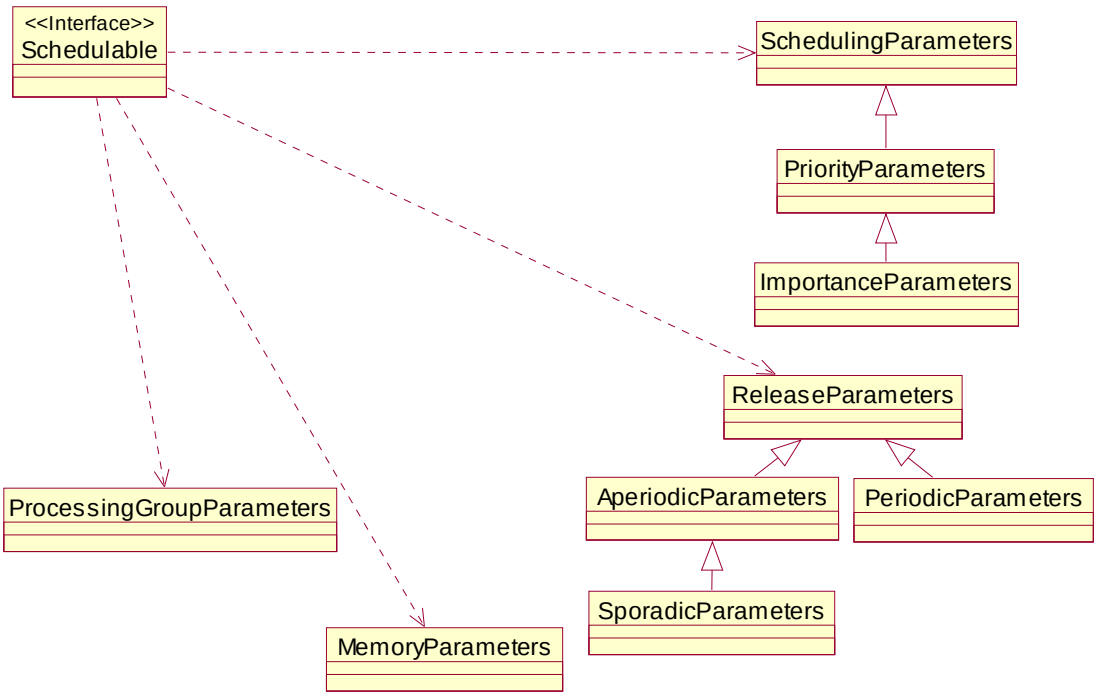


Abbildung 1 Schnittstellenbeschreibung von Schedulable

Es gibt bereits verschiedene Scheduler-Algorithmen, die den unterschiedlichen Anforderungen an die Abarbeitungsreihenfolge und –geschwindigkeit erfüllen. Deshalb besteht die Möglichkeit, statt dem Standard-Scheduler, einen Eigenen mit den gewünschten Merkmalen anzupassen. Dieser Standard-Scheduler ist fixed priority-pre-emptive mit mindestens 28 Prioritätsstufen und ist in RTJ die Klasse *PriorityScheduler*. Diese erbt von *Scheduler*, von der alle Scheduler erben müssen.

In diesem Zusammenhang meint „fixed priority“, dass dem Thread zur Erstellung die Priorität fest zugeordnet und zur gesamten Laufzeit nicht verändert wird. „Preemptiv“ heißt, dass dem Thread, sobald ein laufbereiter Thread mit höherer Priorität verfügbar ist, der Prozessor entzogen wird und der Thread mit der höheren Priorität weiter rechnen darf.

Beispiel[5]:

```
public class ThinkExample {
    public static void run() {
        Object o;
        PriorityParameters pp = new PriorityParameters(10);
```

```
        AperiodicParameters ap = new AperiodicParameters(new
            RelativeTime(5000L,0),new RelativeTime(5000L,0),null,null);
        try {
            System.out.println(“ThinkExample");
             PriorityScheduler ns = new PriorityScheduler();
            Scheduler.setDefaultScheduler(ns);
             MemoryParameters memory = new
MemoryParameters(60000L,60000L);
            RealtimeThread think = new RealtimeThread(pp,ap);
            boolean b = ns.setIfFeasible(think, ap, memory);
             if (b) think.start();
        } catch (Exception e) {
           System.out.println("SchedulerSample: exception");
        }
}
```

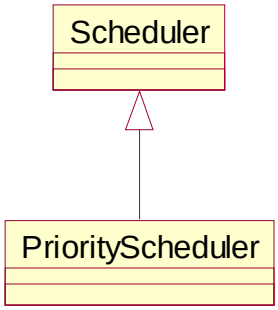


Abbildung 2 Vererbungshierarchie von PriorityScheduler

Um Prioritäten feiner skalieren zu können, wurden 2 weitere Threadtypen eingeführt, bei denen man diese genau definieren kann. Zudem können bei diesen Typen auch noch genau die Speicheranforderungen angegeben werden, um die neuen Speichertypen zu nutzen(siehe Memory Management).

- „*NoHeapRealtimeThread*“: Dieser Thread wird mit der höchsten Priorität ausgeführt und kann nicht mal durch die Garbage Collection unterbrochen werden. Dieser Threadtyp ist also für höchst zeitkritische Threads gedacht. Damit durch die hohe Priorität bei Heapzugriff keine Inkonsistenzen entstehen, darf in diesen Threads der Heap nicht verwendet werden.

- „RealtimeThread“: Die Priorität dieses Thread liegt über der des normalen java.lang.Thread und unter den NoHeapRealtimeThread. Die genaue Zuordnung wird durch die RealtimeParameter getroffen.

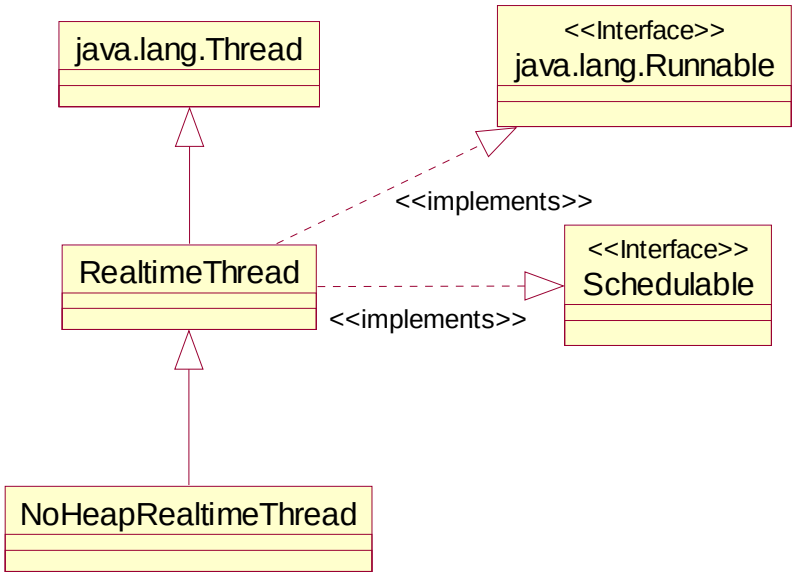


Abbildung 3 Vererbungshierarchie von NoHeapRealtimeThread

Beispiel:

```
import javax.realtime.*;
public class Example1 {
      public static void main(String [] args){
            NoHeapRealtimeThread nhrt = new NoHeapRealtimeThread(){
                  public void run() {
                    System.out.println(“This is a NoHeapRealtimeThread”);
                  }
            };
            RealtimeThread rt = new RealtimeThread(){
                  public void run() {
                    System.out.println(“This is a RealtimeThread”);
                  }
            };
            nhrt.start();
            rt.start();
      }
}
```

3.2 Memory Management

Der Garbage Collector in Java ist zwar eine schöne Sache, weil er sich automatisch darum kümmert, dass nicht benötigter Speicher frei gesetzt wird. Dies kann aber zu einem Hindernis werden, da dieser zum Einen mit nicht vorhersehbarer Zeit die laufenden Threads blockiert. Und zum Anderen Daten löschen kann die, längerfristig von mehreren Threads benötigt werden. Ebenso ist es wünschenswert, dass man mit Java auf

spezielle Speicherbereiche zugreift, um dort beispielsweise direkt Daten von Hardware mit DMA zu bekommen und dorthin zu senden. Deshalb wurde das Memory Management überarbeitet und 4 Speicherarten geschaffen, in denen die Garbage Collection unterschiedlich agieren darf und über die man auch direkt auf physikalische Speicherbereiche zugreifen kann. Für diese Anforderungen, muss die RTJVM direkte Zugriffe auf den Speicher zulassen, wodurch ein großer Teil ihrer Schutzfunktion anderer Anwendungen, vor Java-Anwendungen verloren gehen würde, wenn es nicht die Klasse RealtimeSecurity geben würde. Diese stellt Methoden, die die Anwendungen schützen sollen bereit, doch in der Spezifikation sind keine Details zu der genauen Funktion dieser Befehle genannt.

Folgende Speichertypen wurden neu geschaffen:

- „*scoped Memory*“: In dieser Speicherart wird die Lebenszeit eines Objekts durch die vorhandenen Referenzen auf den/die Threads in diesem Speicher begrenzt. Existiert also keine Referenz mehr auf einen Thread in diesem Speicher, wird dieser freigegeben.
- „*physical Memory*“: Dieser Speicher wird für Objekte verwendet, die Spezielle physikalische Speicherbereiche benötigen, wie z. B. einen DMA-Adressbereich.
- „*immortal Memory*“: In diesem Speicher werden Objekte abgelegt, die zur gesamten Laufzeit der Anwendung existieren. Hier räumt die Garbage Collection nie auf. Dieser Bereich ist bis zum Ende der Anwendung belegt.
- „*Heap Memory*“: Dieser Speicher entspricht dem Heap. Hier wird die Lebenszeit eines Objekts durch seine Sichtbarkeit begrenzt. Dies ist also die traditionelle Speicherart der JavaVM.

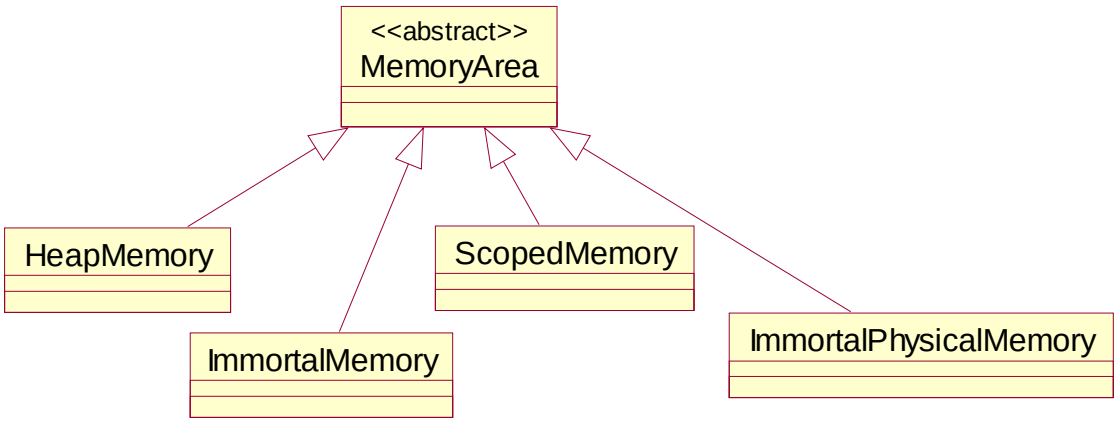


Abbildung 4 Vererbungshierarchie der neuen Memory-Klassen

Ebenso unterstützt das RTJ Memory Management die Reservierung von Speicher für Threads. Für einen individuellen Thread kann der maximale Speicherverbrauch bei der Erstellung angegeben werden.

Beispiel[5]:

```
public class ThinkMemoryExample implements Runnable {
      public void run()  {
            ScopedMemory ma;
            try {
                  ma = new ScopedMemory (65536L);
                  if (!(ma instanceof MemoryArea))
                        throw new Exception("Return object is not
                              instance of MemoryArea");
                  long size = ma.size();
```

```
                if (size < 0)
                        throw new Exception("Memory size is less than_ 0");
            }
            catch (Exception e) {
                System.out.println(“Example: exception");
            }
            try {
                ma.enter(new Runnable() {
                        public void run() {
                                System.out.println(“Thinking”);
                        }
                };
            }
            catch (Exception e) {
                System.out.println("enter(Runnable)_failed");
            }
        }
}
```

3.3 Synchronization

Es wurden zusätzlich zu dem normalen Java Thread zwei zusätzliche Threads, *RealtimeThread* und *NoHeapRealtimeThread* eingeführt. Da diese unterschiedliche Prioritäten haben treten zwangsläufig Probleme bei der Synchronisation zwischen diesen Threads auf. Aber zumindest für die Synchronisation eines normalen Java Threads mit einem NoHeapRealtimeThread werden durch RTJ 3 Queue-Klassen bereitgestellt (*WaitFreeWriteQueue*, *WaitFreeReadQueue*, *WaitFreeDequeue*), die es ermöglichen ohne Verzögerung und Blockierung gemeinsam auf Objekte zuzugreifen. Dies ist aber nicht für die Kommunikation zwischen RealtimeThread und NoHeapRealtimeThread möglich. Hier muss der Programmierer selbst darauf zu achten, oder er sollte diese Konstellation besser vermeiden.

3.4 Asynchronous event handling

In einer Echtzeit-Umgebung muss ein System jederzeit (zu unbestimmten Zeiten und mit unbestimmten Frequenzen oder komplett asynchron) auf ein Ereignis reagieren. Diese Ereignisse können von einem anderen Prozess oder auch von Außen via Hardware–Interrupt ausgelöst werden. Deshalb wurden asynchrone Ereignisbehandlungen eingeführt. In RTJ gibt es 2 Klassen, hierfür:

AsyncEvent steht für ein Ereignis, das ausgelöst werden kann. Dieses kann beispielsweise ein POSIX Signal, ein Hardware Interrupt oder ein berechnetes Ereignis sein. Diese werden mittels „*bindTo()*“ mit den externen Ereignisquellen verbunden. Bei Eintreten eines der verbundenen Ereignisse startet *AsyncEvent* die entsprechenden Behandlungsroutinen der zweiten Klasse(Objekte der Klasse *AsyncEventHandler*), die vorher bereits mittels „addHandler()“ oder „setHandler()“ festgelegt wurden.

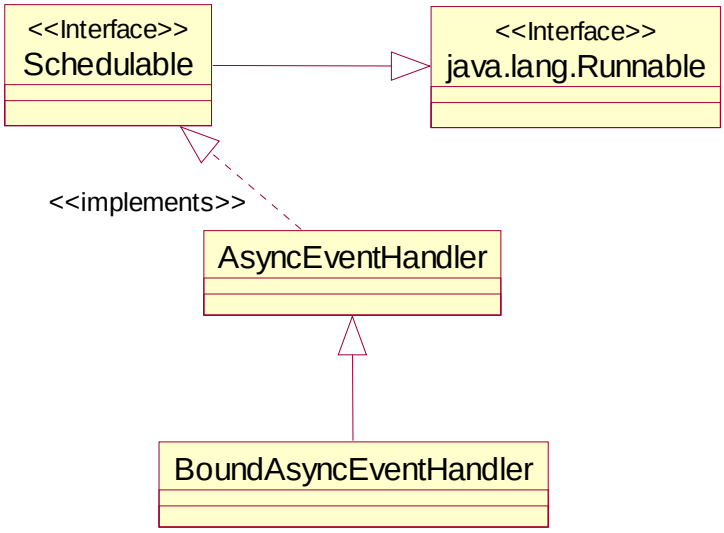


Abbildung 5 Vererbungshierarchie von BoundAsyncEvent

Diese Ereignisbehandlung kann aber auch künstlich durch Aufruf der fire()-Methode stattfinden, die wiederum mit der handleEvent()-Methode des zugehörigen AsyncEventHandler verknüpft ist.

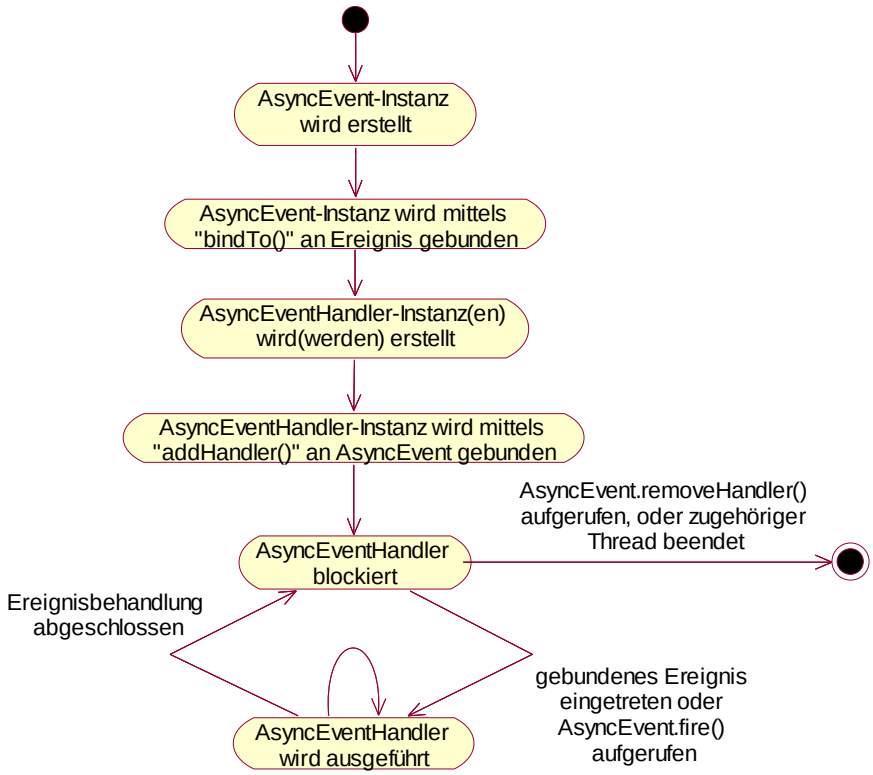


Abbildung 6 Aktivitätsdiagramm zu AsyncEvent

Man kann sie sich wie asynchrone Threads vorstellen, allerdings mit folgenden Unterschieden:

- beim Start werden bei einem AsyncEventHandler gewisse Parameter-Objekte (*ReleaseParameters*, *SchedulingParameters* und *MemoryParameters*) mitgegeben, die das Verhalten bei der Ausführung festlegen.
- AsyncEventHandler sind entweder blockiert, oder werden ausgeführt, im Gegensatz zu Threads, die warten, und schlafen können ([2])
- Threads können eine lange Laufzeit haben, während AsycEventHandler möglichst wenig Code ausführen und terminieren([2])
- Threads müssen unter Umständen auf Ressourcen warten, während AsyncEventHandler möglichst bald ausgeführt werden([2])

3.5 Asynchronous transfer of control

Bei Echtzeitprogrammierung kommt es oft vor, dass Berechnungen nach einer Bestimmten Zeit unter- oder abgebrochen werden müssen, obwohl sie nicht beendet sind. Oder dass die Berechnung mit steigender Anzahl der Iterationen immer genauer wird und man nach einer erst zur Laufzeit bestimmten Zeit, ein möglichst genaues Ergebnis haben will. Dazu wird die Kontrolle dem berechnenden Thread einfach asynchron entzogen. Aber dabei müssen folgende Prinzipien bei RTJ beachtet werden:

- Es muss explizit angegeben werden, dass eine Methode asynchron unterbrochen werden kann, mittels *throws AsynchronouslyInterruptedException*. Würde dies nämlich nicht der Fall sein, käme es bei altem, nicht realtime-fähigen Code zu unbestimmten Zuständen und damit wäre das System gestört.
- Auch in Threads, in denen der asynchrone Kontrolltransfer gestattet ist, kann es Abschnitte geben, die synchron sind und dann natürlich nicht unterbrochen werden können. Es muss jede Methode, die unterbrochen werden darf explizit die throws-Anweisung enthalten. Wenn dies nicht der Fall ist, kann diese Methode nicht unterbrochen werden, auch wenn ihr Aufruf innerhalb einer Methode steht, die unterbrochen werden darf.
- Die Kontrolle kehrt nicht zu der Stelle zurück, an der der asynchrone Kontrolltransfer stattfand. Dies ist auch nicht notwendig, denn dies kann aber durch den *AsyncEventHandler* erreicht werden.
- Der asynchrone Kontrolltransfer wird durch Auslösen einer *AsynchronouslyInterruptedException* erreicht, was durch den Methodenaufruf *interrupt()* auf einen *RealtimeThread* oder *fire()* auf die *AsynchronouslyInterruptedException.*
- Durch den asynchronen Kontrolltransfer ist es möglich einen sichereren Threadabbruch zu haben, wie mit der *stop()* oder *destroy()* Anweisung. Denn *stop()* ist veraltet und *destroy()* ist unsicher, da es den Speicherbereich nicht wieder frei gibt.
- Bei der Modellierung des asynchronen Kontrolltransfer als Exception, muss darauf geachtet werden, dass nur der gewünschte Handler und nicht irgendwelche Universal-Handler angestoßen werden.
- In geschachtelten Threads, die einen Asynchronen Kontrolltransfer erlauben darf der Innere nichts vom Äußeren wissen, aber der Innere darf durch den Äußeren unterbrochen werden, sobald dies der Innere gestattet (Prinzip der Datenkapselung muss erhalten bleiben).

3.6 Asynchronous thread termination

In dem „normalen“ Java ist es schwierig einen Thread „sauber“ zu unterbrechen. Dies ist zum Beispiel dann nötig, wenn sich durch ein äußeres Ereignis die Parameter für den gerade in Berechnung befindlichen Thread geändert haben und diese Berechnung nun so überflüssig ist. Es gab den Befehl *stop()*, aber dieser konnte bei Ausführung Objekte im Speicher zurück lassen. Der andere Befehl *destroy()* kann hingegen zu Deadlocks führen, wenn gerade dieser Thread alle anderen blockiert. Aber durch die RTJ-Erweiterung kann man den Thread bei einem asynchronen Ereignis mittels *interrupt()* unterbrechen und dann bei der Rückkehr zu dem *catch*-Block den Thread sauber beenden.

3.7 Physical memory access

Der physikalische Speicherzugriff kann, wie oben beschrieben, zum Zweck des Zugriffs auf DMA-Bereiche verwendet werden. Es besteht aber auch die Möglichkeit, einfache Datentypen im RawMemory abzulegen. Dazu muss aber beachtet werden, dass diese zum auslesen wieder in den jeweiligen Typ gecastet werden müssen. Durch diese Art des Speicherzugriffs ist man auch in der Lage, Treiber in Java zu schreiben, oder auch spezielle Arten von Speicher anzusprechen, z.B. Batterie-Gepufferten-Speicher, ROM, oder Flash-Speicher. Die Byte-Order wird automatisch aus den Umgebungsvariablen verwendet.

3.8 Exceptions

Da die Echtzeit-Programmierung wieder neuartige Ausnahmen hervorbringen kann, wurden für die Real Time-Erweiterungen einige neue Exceptions geschaffen, wie OffsetOutOfBoundsException, aber da diese nur für die Echtzeitsteuerung vorgesehen sind sollten sie nicht so einfach, wie durch „catch Exception e“ abgefangen werden, man sollte also darauf achten, dass im alten Java-Code nur die Exceptions abgefangen werden, die nur dort auftreten können und nicht aus irgendwelchen Gründen diese neuen Exceptions, die dann unbeabsichtigter Weise falsch behandelt werden könnten. Eine Ausnahme bildet die Asynchronously-InterruptedException, die gehört zu den *java.lang.interruptedException*s und muss dann auch dementsprechend abgefangen werden.

3.9 Time und Timers

Bei Echtzeit-Anwendungen steht, wie der Name schon sagt, die Zeit auch im Vordergrund. Die Zeit aus dem normalen Java *java.util.Date* ist unzureichend, da sie nicht genau genug ist und einige Funktionen fehlen.

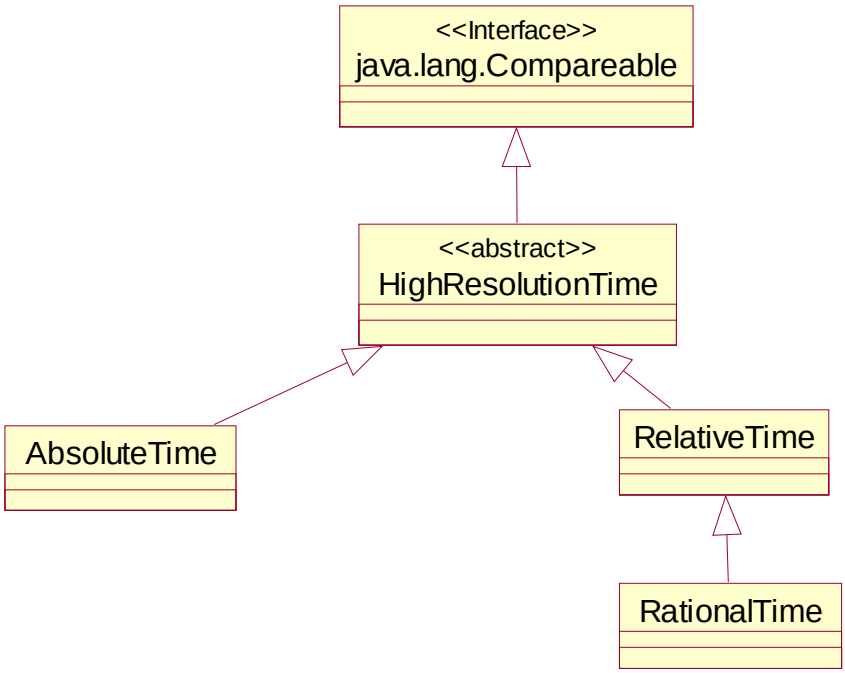


Abbildung 7 Vererbungshierarchie der HigResolutionTime-Klassen

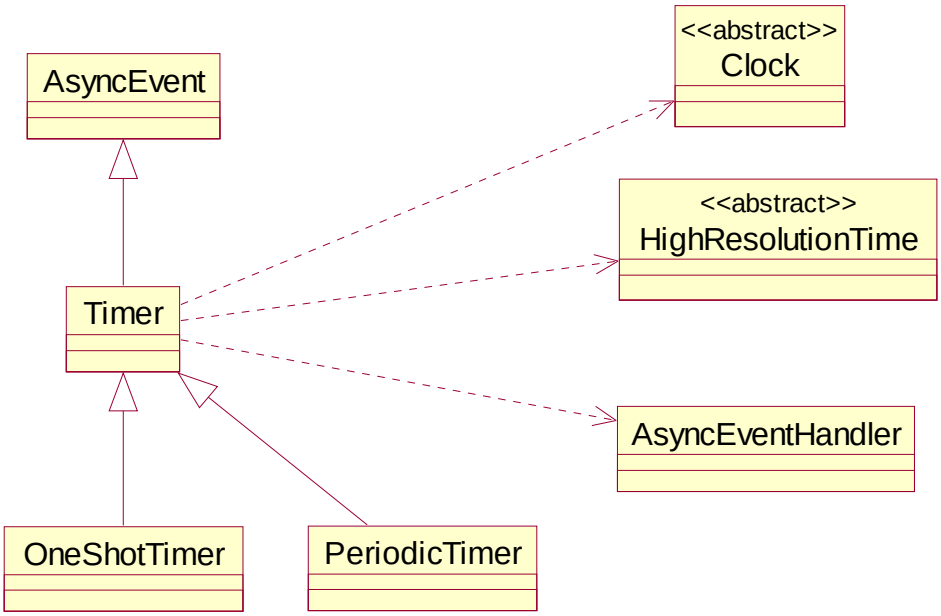


Abbildung 8 Vererbungshierarchie der Timer-Klassen

Deshalb wurde die HighResolutionTime eingeführt. Diese kann die die Zeit in Nanosekunden seit dem 01.01.1970 00:00 erfassen und wird dem entsprechend auch weit genug in die Zukunft reichen. Hierzu werden in der Spezifikation zu Realtime Java leider keine Angaben gemacht. Diese Klasse wird aber nicht direkt verwendet, sondern es erben von ihr nur andere Zeitdarstellungen:

- *AbsoluteTime*: Für die Speicherung der absoluten Zeit seit dem 01.01.1970
- *RealativeTime*: Für die Speicherung einer Zeitspanne
- *RationalTime*: Für die Speicherung einer Frequenz

Diese Unterteilung ist für die Charakterisierung von Threadverhalten gedacht. Denn durch diese Zeittypen kann festgelegt werden in welchem zeitlichen Zusammenhang Threads gestartet werden, d.h. ein Thread soll periodisch aufgerufen werden oder zu einem bestimmten Zeitpunkt oder ähnliches.

Die Timer arbeiten natürlich auf der Basis dieser genauen Zeit. Die Klasse *Clock* gibt die die Zeitbasis und die Auflösung der „Ticks“ vor, sie wird von jedem Timer benötigt. Es gibt zwei verschiedene Timer, zum einen den *OneShotTimer* und zum anderen den *PeriodicTimer*.

Der *OneShotTimer* löst zu einem bestimmten Zeitpunkt ein *AsyncEvent* aus und der *PeriodicTimer* immer in bestimmten Intervallen.

Beispiel:

```
import javax.realtime.*;

public class OSTimer {
     static boolean stopLooping = false;
     public static void main( String [] args)   {
          AsyncEventHandler handler = new AsyncEventHandler() {
               public void handleAsyncEvent() {
                    stopLooping = true;
               }
          };
```

```
            OneShotTimer timer = new OneShotTimer(
                  new RelativeTime( 10000, 0), handler);
            timer.start();
            while(!stopLooping){
                  System.out.println("Running");
                  try {
                        Thread.sleep( 1000 );
                  } catch ( Exception e ){}
            }
            System.exit(0);
      }
}

import javax.realtime.*;

public class PTimer{
      public static void main( String [] args ){
            AsyncEventHandler handler = new AsyncEventHandler(){
                  public void handleAsyncEvent(){
                        System.out.println("tick");
                  }
            };
       PeriodicTimer timer = new PeriodicTimer(
            null, // Start now
            new RelativeTime(1500, 0),  // Tick every 1.5 seconds
            handler);

       timer.start();
       try{
            Thread.sleep(20000);  // Run for 20 sesconds
       } catch (Exception e){}
       timer.removeHandler(handler);
       System.exit(0);
      }
}
```

4 Fazit

Diese Erweiterungen sorgen dafür, dass Java als Echtzeit Programmiersprache verwendet werden kann. Die ersten Projekte mit Real Time Java laufen bereits, obwohl noch nicht der volle Funktionsumfang in die RTJVM integriert ist. Zudem ist, wie bereits oben genannt, die Spezifikation nicht abgeschlossen, weshalb RTJ auch nicht auf so breiter Ebene verwendet wird.

Ebenso ist es noch ein Problem, dass für die bisher erhältlichen RTJVMs nicht einheitlich arbeiten, weil die Spezifikation nur die Sprache selbst beschreibt und nicht die Umsetzung der Sprachelemente auf das Betriebssystems, insbesondere, wie schnell System reagieren soll. Deshalb werden bei einigen RTJVMs noch keine harten Echtzeitanforderungen erfüllbar sein, wie bei einem Test einer RTVM auf Real Time Linux KURT festgestellt wurde[4]. Hinzu kommt, dass zwar von mehreren Firmen an Real Time Java Virtual Machines gearbeitet wird, aber diese nicht frei erhältlich sind.

5 Literatur

[1] Klein, Marcus; Sicherheitskritische Systeme, Real time Linux – Ein Überblick. Siegen, 2004

[2] Eilers, Sönke; Real – Time Java Seminar, Asynchrone Ereignisse und Asynchroner Kontrolltransfer in Real – Time Java. http://parsys.informatik.uni-oldenburg.de/~hybrid/rtjava/rtasync.ppt , Stand: 28.07.2004
[3] Eisma, Aldo; Developing Embedded Systems Using the Java™ Programming Language. http://www.netobjectdays.org/pdf/00/slides/eisma.pdf , Stand: 27.09.2000
[4] Selvarajan, Dinesh; Implementation of Real-Time Java using KURT. http://www.ittc.ku.edu/research/thesis/documents/dinesh_selvarajan.pdf , Stand 08.01.2004
[5] Gu, Lin; The Real-Time Specification for Java. http://www.cs.virginia.edu/~qc9b/cs851/vm-1.ppt , Stand 24.06.2004
[6] Bollella, Greg; Brosgol, Ben; Dibble, Peter; Furr, Steve; Gosling, James; Hardin, David; Turnbull, Mark; Belliardi , Rudy; (DRAFT) The Real-Time Specification for Java.http://www.rtj.org/latest.pdf , Stand: 31.07.2002